Piloteur: A Lightweight Platform for Pilot Studies of Smart Homes

John Feminella
Independently affiliated
jxf@jxf.me

Devika Pisharoty
University of Virginia
dnp6ga@virginia.edu

Kamin Whitehouse
University of Virginia
whitehouse@virginia.edu

Abstract

Despite the comfort and safety that homes provide to humans, they are a hazardous environment for operating embedded sensors and controllers. In this paper, we present the design and implementation of *Piloteur*: a lightweight platform for robust pilot studies of sensing and control systems in the home. Upon installation, it provides end-to-end monitoring of system operation, it repairs problems whenever possible, and it alerts the operator of fatal problems that cannot be repaired. It performs these tasks while minimizing impact on software development time, long-term maintenance risks, and deployment cost. *Piloteur* is an open-source platform offered to the community that consolidates our experience and lessons learned from deploying hundreds of sensors and controllers in homes. In this paper, we present the design and implementation of *Piloteur* as well as our experiences using it to deploy over 180 endpoints across 45 different homes. Despite thousands of potentially fatal node failures, *Piloteur* automatically repaired nodes most of the time and only 26 physical maintenance visits were required over the course of 4 months.

Categories and Subject Descriptors

H.4 [Information Systems Applications]: Miscellaneous

General Terms

Smarthomes, Intelligent Buildings, Software

Keywords

Software, Hardware, Infrastructure

1 Introduction

Sensors and controllers in the home can provide a wide range of benefits and the research community is rapidly developing new solutions that improve comfort, monitor

health, and reduce energy consumption in the home. However, deploying, testing, and evaluating these solutions in actual homes has proven to be an important obstacle to scientific progress in this area. In recent work, we compiled some of the most common causes of system failure and data loss based on our experiences deploying over 1200 sensor deployments in over 20 homes [6], which included disconnections of the home's broadband link, poor wireless connectivity within the home, power outages, snagged wires, and crashed or frozen hardware drivers. Addressing all of these hazards is a costly overhead for any scientific experiment, but failing to address any of these hazards can be even more costly in terms of lost data.

Safeguarding against operational hazards is becoming even more challenging as research in this area matures and pilot studies become increasingly sophisticated. Experimental prototypes can now be built very quickly, sometimes in only a few hours, and so the overhead of ensuring robust and failsafe operation can easily dominate development time. Long-term pilots can last months or years, during which time software can become outdated, online service providers can shut down or change terms of service, and students can graduate. Thus, any dependence on proprietary formats, 3rd party services, or routing maintenance becomes a first class risk for long-term pilots, as well as for archival data that may be kept years or even decades after a deployment is complete. Additionally, pilot studies can now be performed at large-scale, involving dozens or hundreds of houses and devices. These large-scale pilots require a time-efficient installation/configuration process and automated fault detection with a minimal number of false alarms. The financial cost of hardware and software solutions is also a primary concern for large-scale pilots.

Numerous software platforms have already been created to support sensing and control in buildings, but these systems do not fully address the hazards of in-home deployment. Most existing systems aim to provide simplified programming abstractions [2] or advanced features such as data sharing between researchers [9] or running multiple experiments on a single building [3]. These systems either provide no support for operational hazards such as hardware failures, driver failures, and network failures, or they only provide support when drivers, code and data are deeply integrated with the framework, i.e. are built against a pre-specified API, SDK, and/or data abstraction. However, experimental proto-

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the owner/author(s).

BuildSys '14 Memphis, Tennessee USA
ACM 978-1-4503-3143-9/14/11.
<http://dx.doi.org/10.1145/2676061.2674076>

types are often patched together quickly based on 3rd-party hardware drivers and are difficult to force into pre-defined architectures or coding patterns. Yet, no tools currently support arbitrary 3rd-party hardware drivers and the arbitrary data formats that they create. As long-term and large-scale pilot studies of early-stage prototypes become more common, there is a growing need to address operational hazards with a *platform* that provides OS-level abstractions that support arbitrary executables and file formats, rather than with a *software framework* that supports customized code through an API, SDK and data model.

In this paper, we present *Piloteur*: a lightweight platform to mitigate operational hazards for sensing and control systems in the home. *Piloteur* is user-space software that is installed on servers on the Internet as well as on one or more endpoint machines in the home that physically interfaces with sensors and controllers. *Piloteur* does not require that hardware drivers be built against an API or SDK or that data conform to any pre-defined data model. Instead, the interface is defined in terms of basic OS abstractions: executables, files, directories, and utilities. Once installed, *Piloteur* provides the user with three “magic directories” on the endpoint: a *software directory*, a *data directory*, and a *logs directory*. Any software that is put into the software directory is guaranteed to be executed and restarted if it crashes, while any files that are put into the data or logs directories are guaranteed to be mirrored on the servers. Behind the scenes, the *Piloteur* endpoint monitors its network connectivity to the servers and repairs the network when possible, it continuously checks for software updates, it uploads logs of local system state for remote monitoring, and it maintains a reverse SSH tunnel to the servers in case manual maintenance is needed. Meanwhile, the *Piloteur* servers continuously monitor the endpoints’ data logs and operational logs, provides a RESTful Web interface to monitor the status of the endpoints, and triggers alerts to the user if a critical problem is detected.

Piloteur is designed for all stages in the lifetime of a pilot study, including development, setup, deployment, maintenance, and long-term archiving of data after the study is complete. The user interface is designed to minimize the time investment required to use *Piloteur*, beyond the cost of simply building a prototype. A configuration service helps scale deployments up to dozens or hundreds of nodes with only marginal increase in setup and configuration time, and it can run on low-cost embedded devices such as the Raspberry PI for cost-effective scaling. During the deployment, *Piloteur* indicates which endpoints are fully operational, which need remote maintenance, and which need physical intervention. In terms of maintenance, *Piloteur* relies on only basic services that are already readily available on many machines: cron, rsync, SSH, and a Web server. It eschews specialized server processes or 3rd party services that might increase maintenance risks. As a result, *Piloteur* requires little maintenance beyond standard OS updates. *Piloteur* also eschews reliance on proprietary data formats or licensed software that has a risk of becoming unavailable in the future, in order to reduce the risk of archival data decay.

The main limitation of *Piloteur* is that, because it operates

entirely on unstructured data, it cannot provide any of the advanced data management or sharing features that other software frameworks provide. Thus, *Piloteur* complements other software frameworks and is not a replacement for them. We envision *Piloteur* being used as a core platform to enable easy and reliable pilot studies that produce durable data archives, while other software frameworks are layered on top of the unstructured data files when their features become necessary.

We demonstrate the efficacy of the *Piloteur* design with a case study in which we deployed over 180 endpoints across 45 homes, in some homes for about 4 months at the time of writing. Our analysis indicates that essentially all endpoints experience potentially fatal failures that would require manual intervention, but most of the time the *Piloteur* safeguards revived nodes automatically. Despite thousands of potentially fatal software failures, only 26 physical maintenance visits were required to keep over 75% of the nodes operational. Our primary contribution is a lightweight design that enables numerous hazard mitigation techniques to be consolidated into a single platform that meets the needs of a wide range of pilot studies, at multiple stages in the lifetime of a pilot study. Our second contribution is an open-sourced example of this design¹ that can be copied, extended, or re-implemented as necessary.

2 Requirements of Pilot Studies

Sensing and control devices have been installed in homes for many years by home automation enthusiasts, but they generally install only a small number of devices, and typically install only in their own homes. Thus, any operational hazards or device failures can be managed by the homeowner. In contrast, research in smart homes often requires deploying more devices in a single home than a single person can manage, and/or deploying in more homes simultaneously than a single research group can manage. This problem is exacerbated by the fact that, in contrast to commercial buildings that typically use enterprise-grade devices, the home automation market offers primarily consumer-grade products that were not designed for massive scale, remote management, and touchless operation. Using a consumer-grade product for a scientific study requires, among other things, adding safeguards against countless operational hazards: restart the user-level drivers if they crash, reboot the operating system if a kernel-level driver crashes, turn on the machine after a power outage, backup the data files in case of a storage failure, monitor the data files for signs of failure or data loss, alert the user if maintenance is necessary, and so on. Furthermore, residential buildings typically have no power or network provisioning for device installation and have no dedicated facilities managers. As a result, there is a growing need to automatically manage deployments of devices in homes. Based on our own experience, we have observed at least 3 classes of pilot studies that present specific requirements on any platform designed to manage these deployments, described below.

Early-Stage Pilots are used for in-situ testing of an experimental prototype. In our research group, for example, we sometimes buy several comparable off-the-shelf devices (e.g.

¹accessible from <https://github.com/piloteur>

wireless motion sensors) and rapidly test all of them simultaneously in order to choose the best one for a specific task. Even early stage decisions such as this one benefit from pilot tests in a handful of homes for a realistic in-situ assessment. Because these experimental systems have a high chance of being discarded, however, researchers will invest minimal time to make the prototype robust. At the same time, it is difficult to build a prototype using only pre-tested parts, systems, and methodologies because devices for home use often use proprietary communication protocols and require specialized drivers (unlike commercial buildings where devices typically speak a common language such as BACNet or Modbus). Drivers are rarely written from scratch and using a downloaded hardware driver often entails using a specific OS, programming language, and/or development tool chain. The hardware configuration is also constrained by the physical communication port: serial, USB, rs485, Ethernet, and so on. The resulting prototype is often a patchwork of moving parts that does not conform to any pre-defined architecture or design pattern and therefore cannot easily be integrated with an existing software API, SDK, or database schema, as required by many existing tools.

Long-term pilots last months or years and collect data about realistic household activity or energy use. For example, a study may require a full year to capture effects of seasonal weather changes or to observe enough electricity usage for meaningful analysis. During this time, software systems such as databases can require upgrades, sometimes requiring a complex set of upgrade dependencies to be managed. Software licenses and cloud services can change in price or terms of service, or may cease to operate altogether. Thus, a lack of perpetual maintenance can actually be fatal to certain system designs, but scientific pilot studies often don't have a dedicated system administrator and a graduate student who built and deployed a system may even graduate before the pilot study ends. These maintenance risks remain well after the study ends, even for short-term pilots. For example, after 10 years, an archived database may be nearly impossible to revive, or data may be permanently deleted if a cloud storage subscription lapses temporarily. Thus, any platform designed to ensure the long-term integrity of a pilot study must avoid unnecessary dependence on software licenses, 3rd party support, proprietary data formats, cloud services, and routing maintenance: it cannot mitigate short-term operational risks such as power outages and crashed hardware drivers by introducing equally fatal long-term risks due to new software, services, or maintenance routines.

Large-scale pilots may involve dozens or hundreds of houses and devices. For example, the study of a system that depends on human behavior or on a house's floor plan may need to be deployed in a wide variety of households to capture realistic levels of variance. At large scale, however, it becomes unmanageable to manually create safeguards for all these device and/or to manually check whether all systems are operational. Any platform designed to ensure operation of a large-scale pilot studies must therefore include automation support for installation and configuration of safeguards as well as monitoring of system operation. Addition-

ally, the financial cost of hardware becomes a salient factor in large-scale pilots, sometimes limiting the size of the study. Thus, any platform designed for large-scale pilot studies must avoid unnecessary increases in hardware or software costs.

3 Related Solutions

Despite the development of numerous hardware and software tools from both industry and academia, there is still no platform that simultaneously meets the needs of all three classes of pilot studies described above. Existing software generally fall into three categories: data management systems, deployment frameworks, and operational tools.

Data Management Systems collect, store, and provide an interface to a building's data and control points. Most data management systems today are commercial offerings called Building Automation Systems (BAS) or Building Management Systems (BMS) that target facilities managers who need to configure or inspect a building's data points in real time. More recently, several new data management systems have emerged that target building researchers, with a stronger focus on historical analysis and data sharing. For example, SensorAndrew [8] and sMAP [2] provide unified Web-based interfaces to all devices inside a building for simplified application development. BuildingDepot [9] extends this work by providing increased support for fine-grained user access controls. Data management systems usually run on a server or set of servers but sensing and control devices are not managed directly by the system, and they therefore cannot mitigate many operational hazards in the home. For example, a data management system cannot automatically restart crashed hardware drivers, repair failing network connections, or manage the buffering and backup of operation logs. Additionally, most data management systems require the use of a live database, which increases long-term maintenance risk as described above.

Deployment Frameworks operate on devices in the home that physically interface with the sensors and controllers, and therefore have more control over things like hardware drivers and network connections. Recently, a large number of hubs for the "connected home" have emerged on the market, such as the Rovolv Hub and the Staples Connect, that include 6 or more radios and a variety of physical ports to allow multiple diverse devices to all be connected via a single smart phone app. These devices provide seamless network connectivity but do not attempt to guarantee continuous end-to-end operation. For example, they provide no local processing or data buffering and therefore cannot operate in the face of network failures. The Vera made by MiCasaVerde goes one step further in that it provides a programming framework for users to execute their own hardware drivers and apps, but safeguards against operational hazards must be created by the user. HomeOS [3] is perhaps the most sophisticated deployment framework and specifically targets home installations. HomeOS is user-space software that is installed on a machine in the home and, when combined with the Lab of Things [1], provides cloud integration including cloud-based data storage, remote access, and remote management to devices in the home. A key goal of HomeOS is to allow

multiple researchers to simultaneously run different applications on the same home (conceptually analogous to multi-processing) so that home deployments can be shared among researchers and the cost of scaling up large-scale home studies can be reduced. HomeOS can provide robust and continuous operation if code and data are executed/stored within its framework, which includes an SDK, development tool chain, and proprietary data format [5]. External software and data formats can interact with HomeOS but are not offered the same support in terms of operational safeguards, data management, or data sharing. Additionally, the cost of a HomeOS hub is currently quite expensive, as we discuss later, and can quickly dominate total cost in a large-scale deployment, especially if multiple hubs are deployed in each home.

Operational Tools serve as safeguards against a limited set of operational hazards but are not designed to provide end-to-end operational guarantees. For example, SBS [4] and SMART [7] use data analysis techniques to discover when sensors or controllers are failing, but do not help to prevent failures due to common operational hazards. AlwaysUp, UpStart, and other similar tools run any executable as a Windows Service or Linux daemon, guaranteeing that the process runs constantly and sending reports and alerts to the user about performance, uptime, crashes, and other events. Wicd, wpa_supplicant, and other network managers automatically reconnect WiFi after network drops, but do not repair WiFi driver failures and do not send alerts, for example, when a specific server cannot be reached. Tools such as these can improve operation during a pilot study, but they are not designed to work as a holistic system and therefore cannot make high-level decisions, such as when it is worthwhile to sacrifice data collection by rebooting the node to repair a kernel-level driver failure. Furthermore, all of these tools must be installed and configured manually for each prototype, and possibly altered for each deployment or even each home configuration. This piece-meal configuration process itself creates additional risk for continuous operation. Finally, many of the best operational tools require software licenses and/or continued 3rd party support that increase cost and/or create additional long-term risks.

4 Piloteur: The User's View

The user creates and manages a Piloteur network through a command line utility called *piloteur*. The user must first execute the command *piloteur init*, which will create four different types of nodes: a *configuration node* that stores configuration data for all other nodes in the network, a *sync node* that reliably archives data and log files from the endpoint nodes, a *monitor node* that reads the archived data and log files to identify whether an endpoint node has a problem and alerts the administrator and a *bridge node* that gives the administrator remote access to any endpoint node in case a manual software repair is required. Once these four nodes are created, the administrator can create one or more *endpoint nodes* that are installed inside homes to physically interface with the sensors and controllers. Endpoints can be created by executing the command *piloteur create-endpoint*. Upon creation, each node is given a

unique ID as well as a set of *classes* that specify its functionality. For example, a node might belong to classes *motion-sensor* and *Home409*, which means that it inherits the functionality of both motion sensor endpoints and endpoints in Home 409.

The administrator defines the functionality of each endpoint node and class using two directories that are located on the configuration node: the *drivers directory* that holds the software drivers that will collect data from the sensors, and the *configuration directory* that holds configuration files. The former holds the full collection of software drivers that will be used in this network. The latter holds configuration files, which are JSON files with a list of key-value pairs. Configuration files can indicate what type of software drivers a node should run, a home's WiFi ESSID and passcode, the storage limits of an endpoint, or any other parameter important to its operation. If a configuration file is put in the top-level configuration directory, it applies to all endpoint nodes. Configuration files can be applied to only a certain class of nodes by putting them in a subdirectory with the name that defines the class. This approach

The Piloteur utility creates a node by installing user-level software on the base OS of a computer, and a single computer can be configured to serve as two or more different types of nodes simultaneously. For example, a simple Piloteur network would have one endpoint node in each home and the other four types of nodes installed on a single server in the cloud. However, a Piloteur administrator might choose to deploy the monitor node on a separate machine so that its failures are not correlated with those of the sync node, i.e. if the sync node crashes the monitor node would still be able to send alert messages. Similarly, one might choose to deploy a sync node and monitor node on the endpoint machine inside the home for privacy reasons, so that the sensor data never leaves the home site. Numerous different configurations are possible based on network needs.

Piloteur provides a simple interface on each endpoint in the form of three "magic directories": a *software directory*, a *data directory*, and a *logs directory*. Any executable that is put into the software directory is guaranteed to be executed upon startup, monitored, and restarted if it crashes. Any files that are put into the data or logs directories are guaranteed to be mirrored and monitored on the sync node. These files can contain completely unstructured data. If an executable writes to *stdout* or *stderr*, that text stream is automatically redirected to a file in the data or logs directory, respectively. Executables and unstructured data files were chosen as the user interface for Piloteur because they are a *least common denominator* for any endpoint system: all hardware drivers must be executable programs, and all hardware drivers produce output that can be stored as unstructured data, and could produce error messages. This simple design has two key benefits. First, Piloteur is able to extend operational support where existing deployment platforms do not: to arbitrary 3rd party hardware drivers and to arbitrary data formats. Second, it essentially eliminates the development overhead to use Piloteur because software is *not* required to be built against any API or SDK, and data is *not* required to conform to any pre-defined schema or ontology, or to be sent to any database

Figure 1. The Piloteur architecture has five nodes and four services. Each service connects the endpoint node with one of the other types of nodes.

or data management system. The downside of this design decision, on the other hand, is that Piloteur cannot provide any of the advanced data management or sharing features that other software frameworks provide because the data is completely unstructured. Thus, Piloteur complements other software frameworks and is not a replacement for them. We envision Piloteur being used as a core platform for in-home deployments while other software frameworks are layered on top of the raw data files as their features become necessary.

The user interface to a configuration node is also a set of directories, as described above: the *drivers directory* and the *configuration directory*. Similarly, the sync node also provides an interface as a set of directories that mirror those of the endpoints: a single *data directory* and *logs directory*. Each of these two directories has a subdirectory for each endpoint that is connected to it that contains all of that node's data files and log files, respectively. On the other hand, the monitoring node and bridge node have different kinds of interfaces. The monitoring node provides a RESTful Web interface for each endpoint node that details its software versions and operating status. This same information can be retrieved via the Piloteur command line interface. The bridge node provides an SSH tunnel to directly access the endpoint via a command line interface.

A key design goal of Piloteur is to be lightweight. The Piloteur endpoint can be run on low-end embedded platforms such as the Raspberry PI that are headless, fanless, and cost as little as \$25. The small and noiseless form factor of embedded platforms is essential for convenient and non-intrusive deployment in homes, and their low cost is critical when scaling up to a large number of endpoints. The Piloteur server nodes are also lightweight but in a different sense: they eschew the use of databases, data management systems, and directory services used by other building software platforms because these processes themselves require monitoring and maintenance, and thus create yet another operational hazard. Instead, the Piloteur endpoint mirrors its data on the sync node via a standard SSH daemon, and the monitor node processes the data files and log files using a uniform set of scripts executed periodically by a standard cron daemon. All of the Piloteur nodes combined require only three live processes: a cron daemon, an SSH server,

```
"remoteuser": "my-user",
"remotehost": "data.cs.virginia.edu",
"remote_data_path": "smarthome/data/",
"remote_logs_path": "smarthome/logs/",
"data_path": "~/smarthome/data/",
"logs_path": "~/smarthome/logs/",
"watchdog_path": "~/smarthome/watchdog/",
"logging_modules": [
    "monitor",
    "timesync",
    "watchdog"
],
"keyfile_path": "~/smarthome-remote-key",
"log_format": "[%asctime]-15s] [%levelname)s]
               $ %(name)s: %(message)s",
```

Figure 2. An example configuration file contains general parameters for all nodes in a deployment, including the username and server name, the file locations for data files, log files, and executables, and the location of login credentials.

and a Web server. The two servers are among the most reliable and trusted of all servers and are perhaps the most common server processes to be exposed on outward facing ports. Thus, the Piloteur servers are designed to operate indefinitely with minimal maintenance aside from assuring that the OS is updated and the machine has not crashed – both of which are operations that can now be delegated to a platform-as-a-service cloud provider. The Piloteur nodes themselves can be physical machines or cloud-managed virtual machines. The sync node can store data files on any storage medium such as local disk, networked storage or cloud storage. A user is able to choose the machine and storage medium that best matches the desired cost/performance/risk profile.

5 Piloteur: Under The Hood

In order to provide the simple interface described above, four separate Piloteur subsystems must operate together in a coordinated fashion. The four subsystems are illustrated in Figure 1: each subsystem connects the endpoint node to one of the other four types of nodes. The *configuration service* is the first to be activated: it finds a nodes configuration file on the configuration node, downloads the file, and configures the endpoint and its drivers. It then continuously checks for configuration or software updates. The *watchdog service* continuously monitors both the hardware and software of the endpoint, including hardware drivers, peripheral devices, network connections, and Piloteur subsystems. It also maintains a reverse SSH tunnel to the bridge node, in case manual intervention is required. It records all status information in files in a *logs directory* and, if a problem is detected, corrective action is taken if possible. The *sync service* continuously mirrors files from the data directory and logs directory onto the sync node. It then automatically purges old data or log files from the endpoint that have already been mirrored, in order to maintain sufficient local storage for new data. The *monitoring service* continuously monitors both the data files and operational logs from each endpoint, provides a RESTful Web interface that indicates the status of the endpoints, and sends alert messages to the user if a critical problem is detected. The monitoring service is the only subsystem that does not have any code running directly on the endpoint.

```

network={
  ssid="deploymentEssid"
  psk="piloteur"
  proto=RSN
  key_mgmt=WPA-PSK
  pairwise=CCMP
  auth_alg=OPEN
}

```

Figure 3. A class-level configuration file may contain parameters such as WiFi credentials that could apply to all nodes, a single node, or all nodes in a given house.

```

{
  "loaded_drivers": [
    "open_zwave"
  ]
}

```

Figure 4. The `loaded_drivers` parameter indicates which of the drivers in the drivers directory should be executed and logged.

This ensures that the monitoring service will continue to run even if the endpoint fails. Each of these four subsystems is described in more detail below.

5.1 Configuration Service

The configuration service is used to set up and maintain the Piloteur environment on an endpoint. It ensures that the software dependencies are installed, data and code is downloaded, and that everything is installed in its proper place. The configuration service uses a Python library called Ansible to orchestrate and manage deployments. To allow Piloteur to run on a specific endpoint platform, a *playbook* must be written for it. The playbook tells Ansible what operations, or plays, are supported on that environment. Plays are things like "set up a new endpoint" or "update all existing endpoints". The plays themselves are described in the form of individual tasks, like "add a specific SSH key" or "install a software dependency." A full discussion of Ansible is out of scope for this paper, and general knowledge of Ansible is not necessary unless a new endpoint hardware or software platform is being created. Piloteur currently supports two endpoint platforms: Amazon Web Services (AWS) EC2 instances and Raspberry Pi B embedded computers, but the software framework could be extended to additional platforms such as mobile phones or Windows machines.

When a new Piloteur network is created, the configuration node is given two directories: one to store the hardware drivers used during the deployment and another to store configuration files for all nodes in the deployment. These directories must be populated before a new endpoint can be created. Software drivers are arbitrary executable files. Configuration parameters are specified as key-value pairs specified in JSON format, as illustrated by the example configuration in Figure 2. Any configuration file located in the top-level directory applies to all nodes in the deployment. New "classes" of nodes can be defined by creating subdirectories with the class name. Finally, endpoint-specific configuration files can be defined by creating additional subdirectories with the node's unique ID.

When a node is created, the user specifies its classes in a

linear rank order. In the case of conflict where two classes define different values for the same parameter name, values from lower-ranked classes override values from higher-ranked classes. Any values defined for a specific node ID override any other values. This override policy allows the use of a single configuration file for parameters that are shared among an entire class of endpoints, e.g. all endpoints used in a specific pilot study, or all endpoints that will be connected to a specific type of sensor. A separate configuration file can be defined for parameters that are specific to an individual endpoint, such as its house ID or a WiFi password.

As an example, the configuration file in Figure 2 could be applied to all nodes in a single deployment, indicating that they all use the same directory names, sync node, and username. Each individual node might operate only a subset of the sensors and controllers. Thus, the configuration file in Figure 4 would be stored in a subdirectory with a class name for nodes expected to run the "open_zwave" driver. If each node in the deployment has its own WiFi credentials, the configuration file in Figure 3 could be placed in the node-specific directories. If all nodes use the same WiFi credentials, the file could be placed in the top-level directory. If each house has its own credentials but multiple endpoints, new classes could be defined for each house with its the WiFi configuration file for that house. This approach of defining configuration data for classes of nodes ensures that shared parameters are only recorded (and thus only need to be maintained) in a single location.

When an administrator executes the command to create a new endpoint, the network address of the node must be provided, along with a set of classes and a unique ID for the new endpoint. If not ID is provided, a unique ID is automatically generated with the tool `uuidgen`. Then, the Piloteur command line locates the configuration node and generates a single configuration file by merging all configuration files that apply to the new node's unique ID and classes. It installs the configuration service onto the node, the configuration file, and all software drivers in the configuration node's drivers directory. Once an endpoint is created, the configuration service runs itself periodically to keep the system up-to-date. It periodically updates the hardware drivers and configuration files, as well as the base Piloteur system files. If a change is only to a specific driver or set of drivers, those drivers are gracefully terminated and then restarted. If the change is to anything else, all active drivers are gracefully terminated. The configuration service will also roll the system back if it the watchdog or monitoring service (described below) detects that the new configuration causes an operational failure.

5.2 Watchdog Service

The watchdog service runs on the endpoint node to ensure that the hardware drivers, endpoint hardware, Piloteur services, and the network connection are always operational. For each driver named in the `loaded_drivers` parameter in the endpoint's configuration file, the watchdog will: validate that the driver is located in the software directory, run the driver in its own process, and pass any relevant metadata configured for that driver to the process. It runs the driver such that the stdout and stderr streams are redirected to files in the data and logs directories, respectively. If the driver pro-

cess crashes or fails (e.g. because the driver misbehaves, is poorly programmed, or doesn't handle error conditions that it should be handling) the watchdog will restart it. The watchdog will restart the driver process whenever it believes the driver should be running but isn't, as determined by the presence or absence of a lock file that is created by a tool called `flock` when the process starts and deleted when the process dies. The watchdog checks the lock file once per minute.

The watchdog periodically checks the network connection and responds to failures in an escalating manner to diagnose and resolve the issue. The watchdog checks the network connection by opening a socket to the sync node. If this fails, it diagnoses the network to be in one of four possible failure states:

1. *Remote endpoint failure:* External IP addresses are reachable but the sync node is down.
2. *DNS failure:* External IP addresses are reachable but DNS names are not resolvable.
3. *Connectivity failure:* The network interface is up but no IP address beyond the router is reachable.
4. *Interface failure:* The network interface is down or misbehaving.

In the case of interface failure, the watchdog resets the network interfaces. It also records a "strike" each time a corrective action is taken. When the number of strikes exceeds a threshold defined by a configuration parameter called `network_strikes`, the watchdog reboots the endpoint to reset the kernel-level network drivers. When this happens, every monitored driver is sent the `SIGTERM` signal to give it a chance to exit gracefully. If it doesn't finish within 3 seconds, it is terminated with `SIGKILL`. The watchdog performs a maximum of `max_reboots` reboots for a single network failure, which defaults to 3. If a connection is successful after taking one or more corrective actions, the strike counter is reset to zero. In the case of a long-term sync node failure, the endpoint can be configured with a backup sync node to ensure that it does not run out of local storage. Of course, this can cause inconsistent replicas of the node's archive unless the backup sync node shares the same storage partition as the original sync node.

Additionally, the watchdog periodically generates operational logs of the endpoint platform, including CPU utilization, RAM utilization, running processes, status of hardware peripherals, core temperature, and kernel failure or hardware reboots, CPU utilization, disk usage, RAM usage, and so on. If any network failure is detected, a traceroute to the sync node is logged once per failure. Instead of using the Network Time Protocol (NTP) which creates unrecorded shifts in system time, the watchdog periodically logs the local system clock simultaneously with the sync node's system clock so that post-facto time synchronization can be performed. All of these logs are stored in the endpoint node's logs directory, which is automatically mirrored on the sync node. Thus, platform problems can be diagnosed either on the endpoint or the monitoring node. When the network connection is operational, the watchdog ensures that a reverse SSH tunnel is opened to the bridge node. If a software driver detects that

a hardware peripheral that it expects is missing or malfunctioning, it can request a system reboot to repair a suspected kernel-level driver failure.

5.3 Sync Service

The Sync Service is the simplest of the four Piloteur services: it ensures that all data files and log files on the endpoint are backed up on the sync node, and that the local storage on the endpoint is not used to full capacity. The sync service periodically mirrors all files in the magic directories of an endpoint into a directory on the sync node that is named with the endpoint's unique ID. The sync service uses the `rsync` tool to take advantage of differential backups, and runs over SSH so that an additional `rsync` daemon is not required on the sync node (to reduce maintenance risk). The sync service also uses a configurable flush protocol to purge old files from the local storage. It keeps at most `prune_old` files in any given data or log directory. By default, `prune_old` is set to 2 so that the most recent complete log file is always in local storage, along with the log file that is currently being updated. This parameter can be set in the endpoint's configuration files. Other flush policies can be defined to prioritize given directories, to keep files until they reach a certain age, and so on.

5.4 Monitoring service

While the watchdog service attempts to diagnose problems that require autonomous corrective actions by the endpoint itself, the monitoring service attempts to recognize failures that require manual intervention and, when detected, it sends an alert message to the Piloteur administrator. It runs entirely on the monitoring node so that a failure of the endpoint does not prevent monitoring and alerting. The monitoring service infers endpoint state from the data files and log files, and direct access to the node state is not required. For instance, it does not need to check an endpoint's network connectivity directly; if it observes that no operational logs have been synced recently then it infers that either the node or the network is down.

The monitoring service is a Flask Web application that accepts the name of a endpoint, processes the most recent log files, and returns the node's status. It will derive the following status messages, among others:

1. for each driver, the last time that driver wrote anything
2. whether Ansible and the endpoint's dependencies are up to date
3. whether the SSH tunnel is alive and active
4. whether the endpoint's configuration is parseable and working correctly

For each of those items, the monitor gives a color status to each of these items: "green" (fully operational), "yellow" (needs remote repair), or "red" (needs manual repair). The overall status of a endpoint is also reported as a color.

- red: needs manual repair; no network connection or failing hardware.
- yellow: needs remote repair; network connection is up and all hardware peripherals are working, but there is a software error.

Figure 5. No nodes were fully operational for long periods of time (lighter colors indicate more uptime for a node on a given day). Some nodes improved near the maintenance visits (red lines) but all other improvements are due to repairs made through or by Piloteur.

- green: does not need repair; end-to-end data delivery is operational.

When the node is being deployed by a non-expert, the color status can help quickly decide whether an endpoint should be deployed or returned to the lab.

6 Case Study

We used Piloteur to do two pilot studies: an early-stage pilot in 6 homes and a large-scale pilot in 39 homes. Our early-stage pilot was similar to how many other pilot studies are performed today: the pilot was deployed by researchers into the homes of researchers or close friends so that physical maintenance could easily be performed if necessary, and the pilot lasted 2-4 weeks in each home. Our large-scale pilot, on the other hand, took place in the homes of volunteers who were completely unrelated to the research group, and who were recruited over the course of 2 months through the use of flyers and advertisements. Furthermore, the large-scale pilot was deployed by a third-party licensed and bonded installer with no expertise in our software or system architecture. These two aspects of the pilot made physical repairs an absolute last resort and motivated the need for a platform that provides robust, touchless operation. At the time of writing, the large-scale pilot has operated approximately 4 months.

System Architecture: Both pilot studies included three kinds of sensing and control points: Z-Wave motion sensors, Bluetooth Low-Energy (BLE) tags on objects, and a Nest wireless thermostat. This system required on average at least 4 endpoints per home: 1 endpoint installed in a central location that could achieve wireless connectivity with all Z-Wave devices, and 3 or more endpoints with BLE radios that could detect BLE tags at key locations in the home. Unfortunately, these key locations were not guaranteed to be central locations and so a single endpoint could not be used for both sensing systems. The Z-Wave endpoints ran an open-sourced Z-Wave driver called `open-zwave` with an Aeon Lab ZStick USB controller and the BLE endpoints ran a BLE driver called `hcitool` with an IOGEAR adapter for Bluetooth 4.0. All endpoints inside the home were hosted by a Raspberry Pi model B board that used a USB WiFi adapter to connect to the homes broadband network via a secondary,

pre-configured WiFi access point that we installed. An additional endpoint was also required outside the home (running in the cloud) to communicate with the Nest thermostat. The Nest Thermostat does not support direct, local communication via WiFi and all communication must be routed through a proprietary API hosted on a Nest server. Thus, locating the hardware driver inside the home would cause data loss whenever the home’s network drops. Each home had 5 motion sensors and 3 BLE tags on average while the Nest thermostat serves as both a sensing and control point, for a total 4 endpoints, 9 sensing points, and 1 control point per home. Not all homes in the early-stage deployments included a thermostat. In total 180 endpoints were deployed in 45 homes to operate Zwave, Nest thermostat and BLE sensing and control points for over 16 cumulative weeks.

Our sync node and bridge node were hosted on a Linux machine hosted by the University of Virginia’s Computer Science Department. The server is one of many in a computing cluster managed by full-time staff and, although it does not have any service level agreement for uptime, it is highly likely to come back online within 24-72 hours after going down. The endpoints are designed to buffer data for much longer than this, so this inexpensive yet unreliable solution was acceptable for our needs. The sync and bridge nodes require only marginal resources and can therefore run on a shared, general-purpose computing machine. The sync node stored all data files and log files on a network accessible storage partition, also hosted by the department. The network storage service included periodic snapshots for file backup as well as a complete off-site mirror for disaster recovery. This network storage is automatically mounted on the university’s high performance computing clusters via a high speed local network, which makes the data easily accessible for processing.

In contrast, our monitoring node was run on an Amazon Web Services instance. It could have equally well been run on the same server as the sync node, but we chose to locate the monitoring node in a separate fault domain so we would still receive alerts if the sync node went down. This implementation decision meant that monitoring would be a slower and more costly process, but gave additional assurance given the unreliable nature of our sync node. Our configuration node was the GitHub server, which is an inexpensive solution that provides adequate privacy and is unlikely to disappear during the relatively short deployment. For pilot studies with much longer time frames, we would have considered using other hosting methods. Thus, Piloteur’s flexible architecture allowed us to leverage all available resources in a cost-effective fashion. The overall system architecture used in both deployments is shown in Figure 6.

Quantitative Performance: Immediately after deployment we had a 83% operating rate for the endpoints, i.e. 83% of the endpoints reported data to the sync node. Analysis of the operation logs indicate that, at the time of writing, however, all of the 180 total endpoints deployed experienced one or more potentially fatal failures. The uptime for a subset of nodes is shown in Figure 5 for a subset of the study period, where lighter colors represent more uptime for that node on a

Figure 6. The flexibility of the Pilotour architecture allowed the pilot studies to leverage Raspberry Pi’s, UVa-hosted storage and compute nodes, GitHub, and an Amazon Web Services virtual machine.

given day. It is clear from the graph that no nodes were fully operational for long periods of time. The red lines indicate approximately when maintenance visits were dispatched to some of the homes, ± 2 days. All other improvements in uptime are due to repairs made through or by Pilotour.

All 38 Z-Wave endpoints had at least one hardware driver failure, if not more than one. In total, the Z-Wave driver was restarted 27,647 times across 38 endpoints. About 26 endpoints experienced problems with the Ansible system that is used to auto-update the endpoint. 30 endpoints had configuration errors. 13 endpoints had power failures. 31 endpoints had hardware failures and required hardware to physically be replaced. Thus, over 50% of all endpoints had non-network related failures. In 5015 cases, the hardware drivers we fixed by remotely updating the code, where we updated the driver code to another version. In 30 cases, a node was repaired or revived by remotely updating the configuration files, where we updated account information for the Nest drivers that contained information that was no longer valid. 42 endpoints were manually revived through the reverse SSH tunnel.

On the other hand, several nodes needed physical maintenance. 6 endpoints with Ansible problems had to be manually reinstalled. These 6 nodes were deployed in the early-stage deployment and in fact motivated the need for a reverse SSH tunnel. This maintenance was manageable because it could be performed by researchers in the early-stage pilot. The 31 endpoints with hardware failures were fixed with 26 maintenance visits to 15 homes, i.e. some homes were visited more than once. These failures occurred during the large-scale pilot and the maintenance visits were tedious to schedule, and were usually scheduled several days in advance by a combination of email and phone calls with the study participants. Physical maintenance visits to all failed endpoints would not have been feasible.

All 180 endpoints also had network failures at some point, and the sync service actually failed to connect to the sync node over 50,000 times across all endpoints. Most nodes simply had network drops due to intermittent wireless connectivity in the home and regained the connection shortly after. 15 endpoints lost a network connection for long duration when the house’s broadband connection was dropped. 40 endpoints had poor WiFi signal strength as soon as the nodes were deployed, and eventually lost the network connection permanently. Of these, 10 nodes were physically

moved closer to the access point and 30 endpoints are still down. These 30 endpoints inspired an update to the monitoring node’s Web interface, in which the WiFi signal strength is always logged and displayed on the Web page so that the installer could see how good the connectivity was, not just whether the endpoint had connectivity. This new interface was used in subsequent deployments. Buffering data locally on the endpoint was extremely useful since network failures were common and in some cases rsync failed to sync data to the sync node. In relay-based platforms with no on-board processing or buffering, such as those described in Section 3, such intermittent failures would translate to data loss. Even when a manual repair of the endpoint was required, data was synced manually through the SSH tunnel before any repairs were attempted, to minimize the loss of data.

All nodes running the BLE drivers also had hardware driver failures, and the BLE driver was restarted 64,971 times over all nodes. However, this failure rate is in part due to a poorly written driver that requires the hardware to be periodically reset, knowing that the watchdog would always restart the hardware driver. Thus, this is a case where the Pilotour framework did better than not increasing development time; it actually decreased development time by allowing a pilot study to be performed using hardware drivers that were known to have bugs.

The total cost of the pilot study hardware was approximately \$19,100, of which \$10,360 was for the endpoints. In contrast, comparable headless and fanless FitPC nodes to run HomeOS, which was otherwise our platform of choice, would have cost \$60,000 alone, a 6x increase in cost. This additional cost would have quickly dominated the cost of our deployment and would have limited the number of homes that could be included in the study given the amount of funding available. This was a key driver for the creation of a lightweight platform that could execute on low-cost embedded platforms such as the Raspberry Pi.

Pilot Study Anecdotes: The goals of the pilot studies are beyond the scope of this paper, but our experiences with Pilotour during these studies serve as anecdotes to illustrate how and when its features can be useful, as described below.

Creating 190+ endpoints was time consuming, however the platform made the installation easy, after initial setup to the machine used to deploy the platform onto the Raspberry Pi boards. To setup a Raspberry Pi, we installed Raspbian on the SD cards which took about 20 minutes and the platform was installed in an average of 12 minutes. Taking into account time to assemble the hardware (Pi and its peripherals) and test, each endpoint was ready for deployment in 45 minutes. The real bottleneck was the time required for the OS to install on the SD card and time required to ensure that the hardware was robust for long term deployment.

Installation of the endpoints took only about 15 minutes. The installer connected them to power and installed them securely. The installer then used the web interface to ensure that the hubs were online and working. The installer was given a smart phone to access the web interface. The web interface had a couple of minor problems that slowed down deployment. For example, typing in a long identifier

for an endpoint in the search bar was not convenient on a cell phone and typos would lead to an indication that the endpoint was down. This would lead to the installer thinking that the endpoint was not successfully deployed, even though it may have been. We also noticed that the web interface was slow or would timeout and we sometimes had to refresh the page several times to see the search results. These problems are being fixed in new iterations of the system.

Once a node came online, the health of endpoints was assessed using the *deployment monitoring tool*. If an endpoint was “red”, the installer would not deploy the endpoint and a new endpoint was installed at a later time. If an endpoint was yellow, the node could be left for remote repair at the discretion of the installer.

After deployment we noticed that the account information in the configurations files for one of the drivers was no longer valid and we updated this information in the configuration file remotely. The incorrect account information was preventing the driver from logging data. Config files were successfully updated for about 148 endpoints. At a later time, the sync node began dropping rsync requests and the endpoint code was updated remotely to re-attempt an rsync up to 3 times before it logged a failure. This remote software update functionality was critical since rsync eventually failed close to 50,000 times over all endpoints.

We also noticed that portions of the SD cards in some homes would corrupt over time and in some cases, the endpoints would stop working completely. This motivated us to mount the magic directories as tmpfs. New drivers were added to endpoints using the automatic update feature. There was one complication with this solution: when the network adapter failed, the node had to choose between trying to reboot to fix a possible kernel-level driver crash but losing data in the RAM disk, or waiting for a network connection and running out of disk space. This type of trade off is simple to manage with Piloteur through a single configuration option but would be more complex with the use of off-the-shelf software services that individually collect, sync, and monitor data but cannot communicate with each other.

7 Conclusions

In this paper, we present Piloteur, a low-level platform to support robust and reliable pilot studies using arbitrary drivers and arbitrary data formats. Piloteur is a platform and not a software framework. In other words, in contrast to other existing building software frameworks, a key design goal of Piloteur is that application code and hardware drivers are *not* required to be built against any API or SDK, that data is *not* required to conform to any pre-defined schema or ontology, and that data is *not* required to be sent to any database or data management tool. This design minimizes the learning curve, time investment, and overhead required to use Piloteur to mitigate operational hazards. The developer has complete freedom in designing and building the application and can still use Piloteur at deployment time without writing any new code, even if he/she did not originally design the application for use with Piloteur. The user interface for Piloteur was de-

signed to work with a typical development process for rapid prototyping, testing, and experimentation. Piloteur aims to be “lightweight” in terms of three things: 1) software development overhead 2) installation and maintenance overhead, and 3) system requirements for operation.

We demonstrate the efficacy of the Piloteur design with a case study in which we deployed over 180 endpoints across 45 homes. Our analysis indicates that Piloteur reduced maintenance visits to about 35% of the actual number of failing endpoints. Our primary contribution is a lightweight platform design that meets the needs of a wide range of pilot studies and supports multiple stages in the lifetime of a pilot study. Our second contribution is an open-sourced example of this design that can be copied, extended, or re-implemented as necessary. The current implementation is still a work in progress and is not designed to be a final solution that meets all needs, but rather a reference design to build and improve upon.

8 Acknowledgments

This work is based upon work supported, in part, by the National Science Foundation under Grant EFRI-1038271 and the UVa ESPRIT program. Special thanks to Filippo Valsorda for extensive technical expertise and implementation.

9 References

- [1] A. Brush, E. Filippov, D. Huang, J. Jung, R. Mahajan, F. Martinez, K. Mazhar, A. Phanishayee, A. Samuel, J. Scott, et al. Lab of things: a platform for conducting studies with connected devices in multiple homes. In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, pages 35–38. ACM, 2013.
- [2] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. Culler. smap: a simple measurement and actuation profile for physical information. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, pages 197–210. ACM, 2010.
- [3] C. Dixon, R. Mahajan, S. Agarwal, A. B. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *NSDI*, volume 12, pages 337–352, 2012.
- [4] R. Fontugne, J. Ortiz, N. Tremblay, P. Borgnat, P. Flandrin, K. Fukuda, D. Culler, and H. Esaki. Strip, bind, and search: a method for identifying abnormal energy consumption in buildings. In *Proceedings of the 12th international conference on Information processing in sensor networks*, pages 129–140. ACM, 2013.
- [5] T. Gupta, R. P. Singh, and A. P. J. J. R. Mahajan. Bolt: Data management for connected homes. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 243–256. USENIX, 2014.
- [6] T. W. Hnat, V. Srinivasan, J. Lu, T. I. Sookoor, R. Dawson, J. Stankovic, and K. Whitehouse. The hitchhiker’s guide to successful residential sensing deployments. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, pages 232–245. ACM, 2011.
- [7] K. Kapitanova, E. Hoque, J. A. Stankovic, K. Whitehouse, and S. H. Son. Being smart about failures: assessing repairs in smart homes. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 51–60. ACM, 2012.
- [8] A. Rowe, M. E. Berges, G. Bhatia, E. Goldman, R. Rajkumar, J. H. Garrett, J. M. Moura, and L. Soibelman. Sensor andrew: Large-scale campus-wide sensing and actuation. *IBM Journal of Research and Development*, 55(1.2):6–1, 2011.
- [9] T. Weng, A. Nwokafor, and Y. Agarwal. Buildingdepot 2.0: An integrated management system for building analysis and control. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings*, pages 1–8. ACM, 2013.