An Operating System for the Home

Colin Dixon *(IBM Research)* Ratul Mahajan Sharad Agarwal
A.J. Brush Bongshin Lee Stefan Saroiu Paramvir Bahl

Microsoft Research

Abstract—Network devices for the home such as remotely controllable locks, lights, thermostats, cameras, and motion sensors are now readily available and inexpensive. In theory, this enables scenarios like remotely monitoring cameras from a smartphone or customizing climate control based on occupancy patterns. However, in practice today, such smarthome scenarios are limited to expert hobbyists and the rich because of the high overhead of managing and extending current technology.

We present HomeOS, a platform that bridges this gap by presenting users and developers with a PC-like abstraction for technology in the home. It presents network devices as peripherals with abstract interfaces, enables cross-device tasks via applications written against these interfaces, and gives users a management interface designed for the home environment. HomeOS already has tens of applications and supports a wide range of devices. It has been running in 12 real homes for 4–8 months, and 42 students have built new applications and added support for additional devices independent of our efforts.

1 Introduction

Pop culture, research prototypes and corporate demos have all envisioned a smart, connected home where multiple devices cooperate to cater to users' wishes with little or no effort. For instance, in a home with remotely controllable lights, cameras and locks, it should be easy to automatically adjust lights based on the weather and time of day as well as remotely view who is at the door before unlocking it. But such seamless home-wide tasks are conspicuously absent from the mainstream despite the fact that the needed hardware devices are reasonably priced—wireless lightswitches, door locks, and cameras can each be bought for (US) $50–100.

Studies of technology use in the home help explain the gap between the longstanding vision of connected homes and its reality [7, 8, 15, 22, 34]. They find that it is increasingly difficult for users to manage the growing number of devices in their homes. Further, application software that can compose the functionality of these devices is hard to develop because of extreme heterogeneity across homes, in terms of devices, interconnectivity, and user preferences. Finally, finding hardware and software that is compatible with existing home technology is error prone at best. This is problematic as users prefer to organically add a few devices or applications at a time.

We argue that this state of affairs stems directly from the abstractions that home technology presents to users and developers. There are two prevalent abstractions today: an appliance and a network-of-devices. The appliance abstraction is that of a closed, monolithic system supporting a fixed set of tasks over a fixed set of devices. Commercial security and automation systems [1, 12] and many research efforts [29, 43] present this abstraction. The closed nature of such systems means that end users and third-party developers typically cannot extend them, making it a poor fit for an environment where incremental extensions are desired.

The second abstraction is a decentralized network-of-devices. Interoperability protocols such as DLNA [14], Z-Wave [46] and SpeakEasy [16] provide this abstraction. It is also a poor fit for the home because it provides limited or no support for users to manage their technology or for developers to build portable applications that span multiple devices.

In this paper, we advocate for a PC-like abstraction for technology in the home—all devices in the home appear as peripherals connected to a single logical PC. Users and applications can find, access and manage these devices via a centralized operating system. The operating system also simplifies the development of applications by abstracting differences across devices and homes. Further, it provides a central location to extend the home by adding new devices and applications.

We present an architecture to provide the PC abstraction for home technology and its instantiation in the form of a system called HomeOS. Its design is based on user interviews and feedback from a community of real users and developers. It has been under development for over two years.

HomeOS uses (i) Datalog-based access control and other primitives that simplify the task of managing technology in the home, (ii) protocol-independent services to provide developers with simple abstractions to access devices and (iii) a kernel that is agnostic to the devices to which it provides access, allowing easy incorporation

of new devices and applications. HomeOS runs on a dedicated computer in the home (e.g., the gateway) and does not require any modifications to commodity devices.

Our current prototype supports several device protocols (e.g., Z-Wave and DLNA) and many kinds of devices (e.g., lights, media renderers and door/window sensors). It runs in 12 real homes and 42 students have developed applications using it. These homes run applications varying from getting e-mail notifications with photos when the front or back door is opened at unexpected times, to seamlessly migrating video around the house. Students have built applications ranging from using Kinect cameras to control devices via gestures to personalized, face-recognition-based reminder systems.

The experiences of these users and developers, along with our controlled experiments, help validate the usefulness of the PC abstraction and our design. Users were able to easily manage HomeOS and particularly liked the ability to organically add devices and applications to their deployments. Developers appreciated the ease with which they could implement desired functionality in HomeOS, without worrying about low-level details of devices and protocols. These experiences also point toward avenues for future work where we could not provide a clean PC abstraction. For instance, connectivity to network devices, especially wireless ones, is harder to diagnose than for directly connected PC peripherals.

In summary, we make three main contributions. First, we propose using a PC abstraction for technology in the home to improve manageability and extensibility. Second, we implement this abstraction in HomeOS. While we do not claim the pieces of our design are novel, to our knowledge, their use in addressing the challenges of the home environment is novel. Third, we validate the PC abstraction and our design with both controlled experiments and real users and developers.

2 A new abstraction for home technology

Our proposal to use a PC-like abstraction for technology in the home is motivated by our own recent study of home technology [7] as well as the work of others [8, 15, 22, 34]. We first summarize the challenges uncovered by this work, then explain why existing abstractions for home technology cannot meet those challenges, and finally present the PC abstraction.

2.1 Challenges

Home technology faces three main challenges today.

1. Management Unlike other contexts (e.g., enterprise or ISP networks), the intended administrators are non-expert users. But the management primitives available to users today were originally designed for experts. As a result, most users find them hard to use. Worse, devices often need to be individually managed and each comes with its own interface and semantics, rather than having a single, unified interface for the home.

The management challenge is particularly noteworthy when it comes to security and access control where users are frequently forced to choose between inconvenience and insecurity [7, 22]. When they are unable to easily and securely configure guest access for devices (e.g., printers) on their home networks, they either deny access to guests or completely open up their networks.

2. Application development Users want to compose their devices in various ways [34] and software should be able to do just that, but heterogeneity across homes makes it difficult to develop such application software. We identify four primary sources of heterogeneity.

- *Topology:* Devices are interconnected in different ways across homes. Some homes have a Wi-Fi-only network while others have a mix of Wi-Fi, Ethernet and Z-Wave. Further, some devices use multiple connectivity modes (e.g., smartphones switch between home Wi-Fi and 3G).
- *Devices:* Different devices, even of the same type, support different standards. For example, light switches may use Z-Wave, ZigBee or X10; and TVs use DLNA, UPnP A/V or custom protocols.
- *User control:* Different homes have different requirements as to how activities should occur [22]. Some homes want the Xbox off after 9 PM and some want security cameras to record only at night.
- *Coordination:* If multiple tasks are running, simultaneous accesses to devices will inevitably arise. Such accesses may be undesirable. For instance, a climate control application may want the window open when a security application wants it closed.

3. Incremental growth Users frequently want to grow their technology incrementally, as their preferences evolve [7, 22]. Such growth is difficult today because users cannot tell if a given piece of technology will be compatible with what they currently have. This difficulty corners them into buying from one vendor (creating lock-in), seeking expensive professional help, and making significant upfront investments (e.g., buying a home-wide automation system with many features before knowing which features fit their lifestyle). Supporting incremental growth is further complicated by the rapid innovation in hardware and software; users' existing systems frequently do not support these new technologies.

2.2 Prevalent abstractions

Today, home technology can be seen as presenting one of two abstractions to users and developers. The first is the appliance abstraction that provides the same interface that a monolithic, fixed-function device would. It is used for most home security and home automation systems where the set of devices and tasks are both closed. This has the advantage of offering (potentially) simpler user interfaces and simpler integration across the set of involved devices. However, it inhibits extensibility and application development because integration with third-party devices and software is typically not possible. As a result, the security, audio-video, and automation systems are mutually isolated in many homes [7, 22].

The second abstraction is a network-of-devices, which arises from interoperability protocols offering standardized interfaces to devices. This means that, in theory, applications can remotely control devices and devices can be integrated to accomplish tasks. For instance, DLNA allows some TVs to play media content from a computer. In practice, it leaves too much for users and developers to do for themselves. Users interact with each device's own management interface and application developers must deal with all the sources of heterogeneity mentioned above.

2.3 The PC abstraction

The abstractions prevalent today demonstrate the inherent tension between ease of management on one side and extensibility (for both applications and devices) on the other. The appliance abstraction can provide simple user management (at least for the included devices), but typically does not accommodate new devices and applications. On the other hand, the network-of-devices abstraction readily incorporates new devices, but does not provide the needed support for developing cross-device applications or simple management tools.

We propose to resolve this tension by presenting the abstraction of a PC. Network devices appear as connected peripherals, and tasks over these devices are akin to applications that run on a PC. Users extend their home technology by adding new devices or installing new applications without any guesswork with respect to compatibility. They implement desired access control policies by interacting with the operating system, rather than with individual devices and applications. Finally, applications are written against higher-level APIs, akin to abstract PC driver interfaces, where developers do not have to worry about low-level details of heterogeneous devices and their connectivity. Our proposal is inspired by

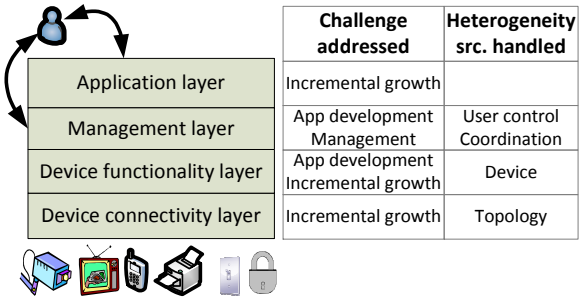

Figure 1: Layers in HomeOS and their considerations

current PC OSes that make some network devices (e.g., printers) appear local. We take this design to its logical extreme, making all network devices appear local, while tackling other challenges of the home environment.

3 HomeOS architecture

HomeOS uses a layered architecture (Figure 1) to bridle the complexity of the home environment and address the challenges mentioned in the previous section. Below, we describe each of the four layers in detail, but briefly, the key elements of our approach are: (i) providing users with management primitives and interfaces that align with how they want to manage and secure their home technology, (ii) providing application developers with high-level APIs that are independent of low-level details of devices, and (iii) having a kernel that is independent of specific devices and their functionality. Our design borrows heavily from traditional OSes but also differs from them in a few key ways.

3.1 Device connectivity layer

The Device Connectivity Layer (DCL) solves the problems of discovering and associating with devices. This includes dealing with issues arising from protocols designed to operate only on one subnet (e.g., UPnP) as well as connecting to devices with multiple connectivity paradigms (e.g., a smartphone on WiFi vs. 3G).

The DCL provides higher layers with handles for exchanging messages with devices, but it attempts to be as thin as possible, avoiding any understanding of device semantics. There is one software module in the DCL for each protocol (e.g., DLNA and Z-Wave). This module is also responsible for device discovery, using protocol-specific methods (e.g., UPnP probes). If it finds an unknown device it passes that up to the management layer where the proper action can be taken.

The DCL frees developers from worrying about some of the most pernicious issues in using distributed hard-

Pan, Tilt and Zoom Camera	DLNA Media Renderer
GetImage() → *bitmap*	**Play**(*uri*)
GetVideo()† → *bitmaps*	**PlayAt**(*uri*, *time*)
Up()	**Stop**()
Down()	**Status**() → *uri*, *time*
Left()	
Right()	Dimmer Switch
ZoomIn()	**Get**()† → *percent*
ZoomOut()	**Set**(*percent*)

Figure 2: Example HomeOS roles and their operations. '†' indicates that the operation can be subscribed to

ware by having a different layer take care of discovering and maintaining connectivity to devices.

3.2 Device functionality layer

The Device Functionality Layer (DFL) takes the handles provided by the DCL and turns them into APIs that developers can easily use. These APIs are services that are independent of device interoperability protocols (§3.2.1), and the DFL is architected to allow easy incorporation of new devices and interfaces whether they are similar to existing ones or not (§3.2.2).

3.2.1 Protocol-independent services

DFL modules provide device functionality to applications using a service abstraction. We refer to service interfaces as *roles*, and each role contains a list operations that can be invoked. For instance, the "lightswitch" role has two operations, "turnOn" and "turnOff," each taking no arguments. Role names are unique with semantics; "lightswitch" implies functionality that is the same across device vendors and homes. Operations may return results immediately and/or allow subscription to events of interest (e.g., when a light switch is physically toggled). Figure 2 shows a few example roles in HomeOS.

In HomeOS, DFL specifications only capture device functionality (and no other detail), and thus the applications that use them do not require changes unless device functionality itself evolves. (We describe below how we handle changing device functionality.) In contrast, the common method today for applications to use network devices is to use a device protocol. For instance, an application might use DLNA to play videos on a remote TV. Using device protocols is problematic because there are many such protocols and they continue to evolve.

We believe that continuous evolution of device protocols is inevitable because they tend to be fat, spanning many layers and concerns. For instance, Z-Wave specifies not only device functionality but also MAC and PHY layer details, and UPnP requires the use of HTTP, SOAP, XML and TCP/IP. As technology changes or new concerns arise (e.g., low-energy, low-bandwidth), new protocols emerge to meet engineering needs.

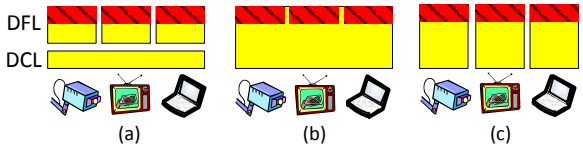

Figure 3: The organization of software modules in HomeOS (a) and two alternative organizations (b) and (c); Solid regions represent protocol-specific code and hatched regions represent protocol-independent code

3.2.2 Extensibility

Introducing new devices in HomeOS is straightforward. A new device can either use an existing role or, if it is a new type of device, a new role can be registered for its functionality. Applications can then be written against this new role, without the need to upgrade HomeOS itself because the kernel is completely agnostic to the services spoken across it. This behavior is different from peripheral APIs in current PC OSes which typically require OS updates to provide common support for new peripherals to application developers. Simplifying the introduction of new functionality is important in the home where new devices arrive frequently. Just in the last five years, devices like depth cameras (Kinect), Internet connected DVRs, and digital photo frames have gone from nearly non-existent to commonplace.

Similarly, if a new capability for an existing device is developed, a new role that exposes the capability can be registered. The DFL module for this device can export both the old and new roles for backward compatibility. Currently, each role in HomeOS is independent; we are considering arranging them in a class hierarchy in which subclasses correspond to more specialized functionality.

3.2.3 Alternative architectures

The HomeOS architecture splits device interactions into two separate layers—a device-agnostic layer (DCL) for basic connectivity and device-specific layer (DFL) that builds on this basic connectivity to mediate between applications and devices. Figure 3(a) illustrates the software organization for three devices that share the same underlying protocol. It is similar to what occurs in PC OSes for USB devices, with a USB-specific module and a device-specific module. However, for network devices, where no universal device protocol exists, this organization may appear ill-advised. The knowledge of a device protocol is spread across two layers, and incorporating a new protocol requires changes to both.

In the home setting, this design has important advantages over alternatives. Figure 3(b) shows one alternative that follows a "one driver per protocol" architecture. It is convenient if there are one or a small number of supported devices in the protocol, but quickly becomes cumbersome to maintain as the number of devices in the protocol grows. A more traditional "one driver per device" architecture produces the software organization in Figure 3(c) where both connectivity and functionality are mediated by a single piece of software for each device. This not only results in DCL functionality being replicated if multiple devices use the same protocol, but also necessitates coordination among modules communicating to different devices. For instance, Z-Wave only allows one live message in the Z-Wave network at a time requiring coordination among any software modules that might send Z-Wave messages. Thus, after experimenting with these alternatives, we chose to split device communication across DCL and DFL as shown in Figure 3(a).

3.3 Management layer

The management layer in HomeOS provides two key functionalities. First, it provides a central place to add and remove applications, devices, and users as well as to specify access control policies. Second, it mediates potentially conflicting accesses to devices ensuring that applications do not need to build their own mechanisms to handle shared devices.

We provide both functionalities using the same primitives. A key goal for their design is that they be simple and translate into management interfaces with which users can easily implement desired policies. Otherwise, we risk not only user frustration but also misconfigurations with serious security and privacy consequences, given the sensitive nature of many devices (e.g., cameras and locks). To gain insights on the design requirements, we conduct a study of households with existing automation. We describe the study in §3.3.1 and the primitives in §3.3.2. While these primitives do not extend the state of the art beyond what researchers have proposed in the past, their simplicity and universal application across an OS goes beyond commodity OSes.

3.3.1 Understanding management requirements

Users have complex needs of home technology. While different than those of experts, their own mental models are often refined. To understand these mental models and common user activities, we visited households with home automation already installed (e.g., remote lighting and locks, security cameras). While we expect HomeOS to enable tasks not possible today, these households can give us insight into their experience with current technology as well as how they would like to manage home technology in the future. We present results on the former elsewhere [7] and focus here on the latter aspect. We interviewed 31 people across 14 homes (1 in the UK, rest in the USA), with different systems such as Elk M1, Control4 and Leviton.

Our visits revealed that households want access control primitives that differ from those present in traditional OSes. We summarize four important differences below.

1. Time-based access control Our participants wanted to control access to devices based on time. Parents mentioned restricting children from using certain devices after certain times (e.g., "If my daughter wanted watch [Curious] George at 11 o'clock at night, I wouldn't want to do that"). While social interaction suffices to address some of these concerns, many parents asked for technical means as well. Time-based access control is also needed to give households an ability to grant a variety of access durations for guests (e.g., a few hours to babysitters and a few days to house guests).

Current commodity OSes provide coarse-grained parental controls that can limit whole accounts to certain times of the day, but they lack flexible controls that can easily implement policies such as those above.

2. Applications as security principals Users highlighted a desire to be able to limit applications' abilities to access devices. One participant said "I don't want to grant it [the application] access to everything, just my laptop." A participant in a different home commented about another application: "if it said my DVR and my TV I would say fine, ... if it had my phone or my computer I would want to be able to choose [what it can access]."

This observation requires treating applications as first-order security principals, in addition to users. In current PC OSes, users alone are the primary security principals (with some exceptions such as firewall rules in Windows), and applications simply inherit users' privileges. While smartphone OSes treat applications as security principals, they are solving the simpler problem of regulating single-user, self-contained resources.

3. Easy-to-understand, queryable settings As expected, users complained about complicated interfaces to configure devices (and especially security), but they also bemoaned the lack of a simple way to verify security settings. They had no way convince themselves that they had correctly configured their settings. For example, to ask if guests can access security devices or if a given application cannot unlock the door after 10 PM.

Providing reliable answers to such questions is diffi-

cult in current OSes due to issues such as dynamic delegation [10]. In the home, the consequences of incorrect configurations can be severe, requiring even more confidence in security. The lack of such a capability can scare users away from the idea of using new or potentially dangerous capabilities, even if it is possible they are correctly configured. For instance, a participant with electronic door locks said he had not hooked up remote access because he was not "100% certain of its security."

4. Extra sensitive devices Our users showed heightened sensitivity for the security and use of certain devices (e.g., locks and cameras). They wanted support to ensure accidentally granting access to such devices was difficult.

3.3.2 Primitives

The requirements for security and access control outlined above are in conflict. The first two call for primitives that are richer than those in current OSes. However, non-experts find it hard to configure and understand even those primitives [10, 33]. We reconcile the conflict by noting that the home is a much simpler environment that does not need much of the complexity motivated by enterprise environments (e.g., dynamic delegation, highly-customizable ACLs and exceptions).

Datalog access control rules We formulate access control policies as Datalog [9] rules of the form (r, g, m, T_s, T_e, d, pri, a), which states that resource r can be accessed by users in group g, using module m, in the time window from T_s to T_e, on day of the week d, with priority pri and access mode a. Time window and day of the week lets users specify policies by which something is allowed, for instance, on Sundays 7–9 PM. Groups such as "kids" and "adults" are configured separately. Priorities are used to resolve conflicting access to the same resource. Access mode is one of "allow" or "ask." With the latter, the users have the option to permit or deny access interactively when the access is attempted. Studies show that users prefer this flexibility rather than having to specify all possible legal accesses a priori [5, 33]. Any access that is not in the rule database is denied. While these rules may seem complex for users at first, they are amenable to visualization and English sentences like "Allow residents to access the living room speakers using the music player from 8 AM to 10 PM."

Expressing access control as Datalog rules meets our requirements. Users can configure time-based policies as well as restrict an application to accessing only certain devices. They can also easily understand their configuration by queries such as "Which applications can access the door?", "Which devices can be accessed after 10 PM?" or "Can a user ever access the back door lock?" to fully understand their risk. Reliably providing such views is straightforward because they can be formulated as Datalog queries. Answering these queries is fast despite there being many dimensions per rule. Because the policies are straightforward, as we show later, even non-experts can configure and understand them.

The main advantage of Datalog over ACLs is its simplicity. ACLs can be more expressive, assuming we extend them to include time and applications. But they are hard for users to program [33] and hard to aggregate and summarize. We are not the first to propose the use of Datalog for access control, but its use can require major extensions to accommodate policies of complex environments [32]. We find that the needs of the home environment can be met without such extensions.

Past systems looking to simplify access control have explored using a simple table [38] with the principals along one axis and the objects along the other with each cell specifying if access should be allowed. While promising, this approach does not scale well beyond two dimensions and our interviews indicated that time and application were both required dimensions.

Time-based user accounts In addition to the use of time in access rules, user accounts in HomeOS can have an associated time window of validity. This window is used to simplify guest access, which studies have shown to be both common and particularly problematic [22, 33]. Home owners can start access for a user at a certain time (e.g., for a future guest) and terminate access at a certain time (e.g., when the guest is expected to leave). The data corresponding to the guest (e.g., access privileges) are not deleted automatically after the validity window, to simplify reinstating access at a later time.

Hierarchical user and device groups Groups in HomeOS are arranged in a tree hierarchy. In contrast, groups in current OSes can be independent sets. We picked the tree organization because of its simplicity. When a user group is given access, it enables an easier determination of which users are given access. A user who is not part of this group will not inadvertently gain access because she is part of another group.

For devices, we use a tree hierarchy that is rooted in space because that matches how users think of resources in the home. It also aligns well with physical access as it is delineated by rooms. To our knowledge, current OSes do not support such device groups. We find that device groups simplify management; users can specify policies for groups rather than individual devices.

Orthogonal to spatial grouping, HomeOS has a high-security group. Users can deem certain devices as high-security to avoid accidental access to such devices and simplify the task of keeping the home network secure.

Applications are not given access to secure devices by default, and the user must explicitly provide access to such devices. We add some common classes of devices (e.g., cameras, locks) automatically to this group; users can later add or remove devices to or from this group.

Access control also forms the basis for privacy in our design. Applications cannot access sensor data unless they are granted access to those devices. Further, network access is disabled by default, so they cannot leak information externally. (Software updates are downloaded and applied by HomeOS.) Thus, we coarsely control privacy at the granularity of applications and devices. In the future, we will consider finer-grained control [42, 44].

3.4 Application layer

The application layer is where developer-written code runs. The key feature this layer provides, beyond the ability to use and compose devices, is the ability to determine if an application is compatible with the home and what services and/or devices are missing if it is not.

Today, users have little assurance that a given piece of software will work in their home. To address this uncertainty, HomeOS requires that applications provide a *manifest* describing what services they need. This enables it to determine if an application will function with the current device services in the home. (A similar approach is being used to manage handset diversity in smartphones today.) If the manifest indicates an application is not compatible, HomeOS can also determine what additional devices or services are needed.

A manifest has mandatory and optional features. Each feature is a set of roles, at least one of which is needed. For instance, an application may specify {"TV", "SonyTV"}, {"MediaServer"} as mandatory features, indicating that it needs a service with at least one of the two TV roles and a service that exports a media server. It might have {"Speaker"} as an optional feature if it offers enhanced functionality with that role.

Our current manifest descriptions cannot encode complex requirements (e.g., if an application needs devices to be in the same room). They handle what we deem to be the common case. Should the need arise, it would be straightforward for us to enhance manifest descriptions.

4 Design and implementation

HomeOS is an implementation of the above architecture as a component-based OS. All functionality that is not central to the platform is implemented by software components called *modules*. Modules that sit in the application layer are *applications*, and those that sit in the DCL and DFL are *drivers*.

4.1 Modules

Modules are the basic unit of functionality in HomeOS and, whether applications or drivers, they implement the API described in Figure 4(a).

Before a module can be run, it must be installed. Driver modules are installed when new devices are discovered on the home network; applications are installed in response to explicit user directives. Modules are installed by copying the binaries and accompanying metadata (e.g., manifests) to a specific directory. Installation is carried out only if the module is deemed compatible with the devices in the home. This check is also performed each time the application is run to deal with configuration changes. During installation, users specify if the module should be started automatically upon system (re)start or only upon explicit user request. Module updates and uninstallation are carried out by HomeOS and not by the module itself.

Running modules are isolated to prevent any interaction except via the APIs to the HomeOS platform and the service interface. By default modules are denied access to the network. DCL modules are the exception as some must use the network to control their associated devices. Even then, when possible we limit connectivity to only those devices. A module's file system access is limited to its own working directory where it can store its data and configuration.

HomeOS relies on DCL modules to discover new devices on the home network by running protocol-specific discovery protocols. (We also support a mode where users can also manually add a device if HomeOS is unable to do so automatically.) Once a new device is discovered, HomeOS installs a DFL module for it based on a database of device type to driver mappings. The device type is reported by the DCL module and is protocol-specific. The DFL module is granted access to the service that the DCL module exports for this device.

4.2 Services

Services are the only way that modules are allowed to interact with each other. They do so using a standardized API described in Figure 4(b). Modules advertise the services they offer to HomeOS which keeps a history of offered services to enable future compatibility testing.

Start: Called to start a module; modules are garbage collected when it returns
Stop: Called to request a module to stop; where state can be cleaned up before exit
SvcRegistered: Called when a new service becomes active; used to listen for services of interest
SvcDeregistered: Called when a service becomes in active; used to avoid using inactive services
AsyncReturn: Called whenever a subscription generates an event or asynchronous call returns

(a) API for HomeOS modules

InitSvcAndCapability: Creates a service and a capability to access it; returns the service handle back
RegisterSvc: Registers the service as active advertising it to other modules
DeregisterSvc: Marks a service as being inactive and notifies other modules
GetAllSvcs: Returns a list of all active services
GetCapbility: Requests a capability to access a given service
IsMySvc: Returns whether a given service belongs to this module
Invoke: Used to call an operation either synchronously or asynchronously
Subscribe: Subscribe to notifications from an operation
SpawnSafeThread: Create new thread which safely propagate its failures

(b) API For finding and interacting with HomeOS services

Figure 4: The APIs for modules (a) and services (b) in HomeOS

Modules inform HomeOS about services of interest to them using role names and/or locations. (Note that expressing interest does not grant access to a service other than to know of its existence and description.) When a service is registered, modules that have expressed interest in it are notified. A module can then query the service for its description as well as its location. Querying for the existence of a service does not require access privileges.

When a module needs to invoke an operation on a service, it requests a capability [31] from HomeOS. As part of the request, the module passes the credentials of the user it is running on behalf of. Without valid credentials, the request is successful only if the access is legal for all users. Drivers are handled slightly differently and typically have rules that give them access to their corresponding devices regardless of the user. User-controlled policy is applied when applications access the driver rather than when the driver accesses the device.

The legality of the requested access is evaluated by HomeOS based on the user, module, service, and time of day, by formulating the check as a Datalog query over the users table and access rules. If legal, a capability is generated and returned. A copy of the capability is then passed to the target service; this makes it easier for HomeOS to revoke the capability later if needed. Subsequently, the requesting module can use the capability to make calls directly to the service. HomeOS capabilities have an expiration time based on access rules.

An operation's callers must also include input parameters of the right type. HomeOS supports both primitive and complex types, which are passed by reference across isolation boundaries to avoid the overhead of serialization. Finally, operation invocations include a timeout. Unless the called service responds within this time limit, a timeout error code is returned.

4.3 HomeStore

To simplify the process of finding new applications and devices, inspired by smartphone app stores, HomeOS is coupled with HomeStore which hosts all HomeOS applications and drivers. It indexes application manifests as well as drivers' associated devices and exported services. It helps users find applications that are compatible with their homes, by matching manifests against services in their home. If an application is not compatible, it can recommend additional devices that meet the requirements.

4.4 Management tasks

To explain how users manage their homes using HomeOS we describe four important management tasks.

1. Adding a new application Users can browse applications and view their compatibility within HomeStore. Upon installing an application, HomeOS walks the user through setting up access control rules for the application. The core of this task is specifying which devices (or services[1]) the application should be allowed to access.

Since there may be hundreds of devices, we use the application manifest and service descriptions to show only compatible, non-secure services. Once the user selects which services the application can access, HomeOS uses the Datalog rule database to detect if the new application could access a device at the same time as other applications. If so, it asks the user which application should have a higher priority.

2. Adding a new device Once a new device is registered, users need to specify its location and whether it should be marked secure. They also need to configure which existing applications should have access to the device. This task is again simplified using application manifests, as only applications compatible with the new device are presented as valid options.

3. Verifying access rules To verify access control configuration, HomeOS allows users to view the rules from different vantage points using faceted browsing [25] (found on shopping Web sites to filter content

[1] HomeOS devices are exposed as services, as are features like notification and face recognition. To ease exposition, we refer to devices.

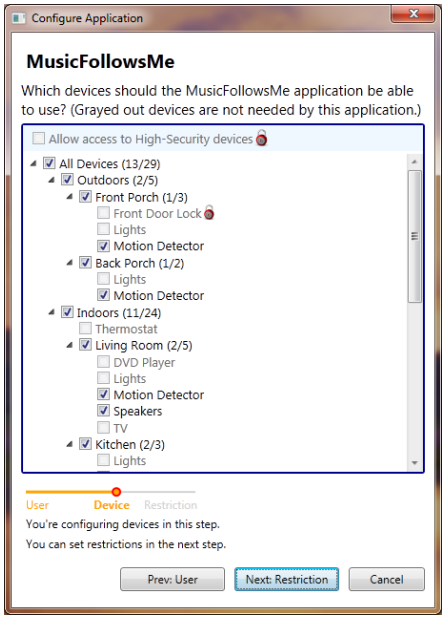

Figure 5: A GUI screen capture showing how applications are given access to devices

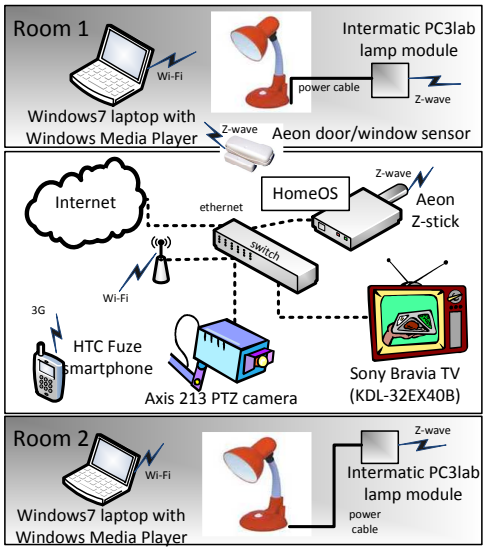

Figure 6: Our implementation testbed

along multiple dimensions). This enables users to pose questions such as what devices an application can access or what devices can be accessed at night by a user group. The questions are answered using Datalog queries.

4. Adding new users When a new user account is added, the administrator must specify their group (e.g., guest) and the time window of account validity.

We have built a complete user interface (UI) to support these tasks. An example screenshot is shown in Figure 5 which occurs during the course of adding applications. We omit detailed description of the UI for space constraints, but note that it closely mirrors our system primitives and includes heuristics designed to minimize the exposure to risk even if users click OK repeatedly during configuration activities. Evaluation of its usability, which we discuss later, also evaluates the manageability of our primitives.

4.5 Implementation

We implemented HomeOS in C# using the .NET 4.0 Framework. We use the *System.AddIn* model which allows dynamic loading and unloading modules. It also offers module version control allowing the HomeOS kernel and individual modules to evolve independently. We isolate modules using *AppDomains*, a lightweight sandboxing mechanism [2]. Each module runs inside a domain. Direct manipulation is not allowed across domains. Instead, communication is done only through typed objects exchanged through defined entry points and subject to access control. As our evaluation shows, the overhead of this isolation mechanism is low enough to support interactive applications. However, it does not provide performance or memory pressure isolation; these are subjects of future work. For Datalog query evaluation, we use Security Policy Assertion Language (SecPAL) [40], after modifying it to support time comparisons.

In addition to the HomeOS kernel, we developed drivers to interact with a diverse set of off-the-shelf devices that includes Z-Wave lighting controls, door/window sensors, smartphones, network cameras, TVs capable of receiving DLNA streams, Windows PCs, and IR (infra-red) transmitters. Figure 6 shows one of our testbeds with some of these devices.

Interoperability protocols greatly simplified the task of making these devices work with HomeOS because a module corresponding to the protocol can communicate with different devices. For instance, the same DLNA module works with both Windows 7 computers and the Sony TV; and the Z-Wave lightswitch module works with both Aeon and Intermatic devices. Greater adoption of interoperability protocols will make it easier to integrate devices with HomeOS. See below, however, for some shortcomings of current interoperability protocols.

We developed 18 applications that use these devices and run on our testbeds. While some applications require access to only one device (e.g., turn on or off a light), others are quite complex. For example, a media application transparently redirects a music stream from one room to another, depending on how lights are turned on or off in each room (using lights as a proxy human presence in the room). We also implemented a two-factor authentication application that triggers a configured action (e.g.,

open a lock) when the same person authenticates with their voice on the phone (speech recognition) and with their face on the camera (face recognition).

Each of the 18 applications is less than 300 lines of C# code and took only a few hours to develop. Because applications are written against high-level abstractions, most of the effort went toward application-specific logic. As we report below, other developers also found application development in HomeOS to be easy.

5 Experience

HomeOS currently runs in 12 real homes and 42 developers have written modules for it. These field experiences validate key aspects of the HomeOS architecture. They also evaluate the utility of the PC-like abstraction for home technology as well as situations in which HomeOS was more or less able to preserve that abstraction. Generally, the experiences were positive even for people with no prior home automation experiences. However, the experiences did reveal some rough edges of our prototype and limitations of current interoperability protocols.

We describe the main findings below, based on informal surveys of our users and developers. The next section presents results from controlled experiments.

5.1 Developers

We gave the HomeOS prototype to ten academic research groups, for use as a platform for both teaching and research on home applications. As part of this program, 42 undergraduate and graduate students developed tens of HomeOS applications and drivers.

They extended HomeOS in several directions [3, 4, 26, 27]. They wrote drivers for new devices including energy meters, different network cameras, appliance controllers and IM communication. They wrote new applications such as energy monitoring, remote surveillance, and reminders based on face recognition. The PC-like abstractions of drivers and applications enabled them to build software quickly and in reusable modules. Moreover, as a testament to the flexibility and extensibility of its architecture, we were not required to—and did not—modify HomeOS to support these development efforts.

As an example, one group extended HomeOS to support the Kinect RGB-D camera and built an application which allowed users to control lights via gestures (Figure 7). They were able to do this without having to wait for Kinect integration with a large commercial system (e.g., Control4) and were able to get it to interact

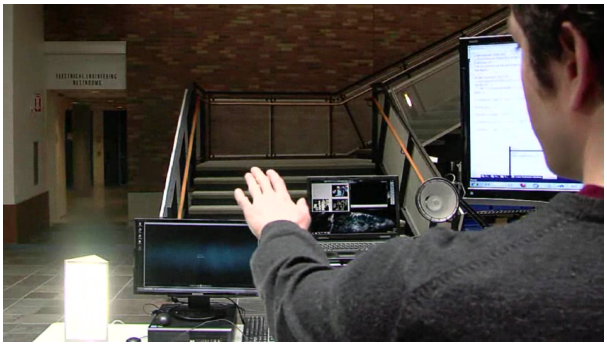

Figure 7: A student demonstrating how to turn on and off lights via gestures with a Kinect

with existing devices. Another group built an application that plays audio reminders based on who is recognized on cameras. This works with webcams, security cameras, Kinect or IP cameras and with any device HomeOS can play audio through (right now PCs, DLNA TVs, and Windows Phones). This underlines the power HomeOS gives to application developers to easily span multiple types of devices (security, PC, phone, entertainment, etc.). Commercial systems today support only a subset of devices related to their target scenario (e.g., security systems focus on cameras and motion sensors).

Layering and programmability Developers who wrote applications found the protocol independence of the APIs appealing. Developers who wrote new drivers for devices with existing DCL modules (e.g., a Z-Wave appliance controller) liked that they did not have to concern themselves with the low-level connectivity details and could instead focus exclusively on device semantics.

Interestingly, developers who extended HomeOS to devices without an existing DCL module (e.g., ENVI energy meters [19]) started by building one module that spanned both the DCL and DFL. For them, the split was unnecessary overhead as only one device used the connectivity protocol. However, in one case a group had to support multiple devices with the same connectivity protocol based on IP to Z-Wave translation. This group found value in separating functionality across two layers indicating it was not just an artifact or our experience.

Hardware-software coupling Our developers sometimes wanted to use device features that were not exposed to third-parties over the network. For instance, one developer wanted to insert a text notification on a TV without otherwise interrupting the on-screen video. Today, some set-top boxes have this capability (e.g., for cable TV operators to signal caller identity of incoming calls), but they do not expose it to third-party software.

This points to an inherent advantage of vertically integrated software—being able to better exploit device capabilities—that open systems like HomeOS lack. This

is unsurprising in retrospect as the closed nature of current solutions and devices is what HomeOS attempts to combat. However, the systematic way vendors can expose device capabilities in HomeOS should encourage them to make their capabilities available to applications.

Media applications and decentralized data plane A few developers had difficulty in writing media applications. HomeOS centralizes the control plane but not the data plane to avoid creating a performance bottleneck. If two devices use the same protocol, we assume that they can directly exchange data. Thus, we assume that a DLNA renderer can get data directly from a DLNA server once provided with a media URI. The DLNA protocol turns out to not guarantee this because of video encoding and/or resolution incompatibilities. While we currently use heuristics to provide compatible formats when transcoding is available, they are not perfect.

For reliable operation, we also plan to use HomeOS as a transcoding relay (thus, centralizing the data plane and more closely mirroring the PC abstraction) when data plane compatibility between nodes is not guaranteed. As high-quality open source transcoders exist [21], the main technical challenge is to generate profiles of what input and output formats devices support. This requires parsing device protocols like DLNA. Although this means violating HomeOS's agnostic kernel, we believe that media applications are common and important enough to justify an exception.

5.2 Users

Twelve homes have been running HomeOS for 4–8 months. We did not actively recruit homes but many approached us after becoming aware of the system. We limited our initial deployment to 12 homes. Ten of them had no prior experience with home automation. Beyond providing the software and documentation, we did not assist users in running or managing HomeOS. These homes are using a range of devices including network cameras, webcams, appliance and light controllers, motion sensors, door-window sensors and media servers and renderers.

Organic growth What our users found most attractive was being able to start small and then expand the system themselves as desired. At first, they typically did not know what they wanted and only discovered what they found valuable over time. HomeOS let them start small (at low cost) and extend as needed. It thus provided a system that was much more approachable than commercial systems today that require thousands of dollars upfront. It was also more likely to satisfy users by allowing them to evolve it to meet their needs rather than requiring them to make all decisions during initial installation [7].

Indeed, all users employed an organic growth strategy. One user started running HomeOS with only one network camera to view his front yard on a smartphone while away from home. He later added two more cameras—a webcam and a network camera from a different vendor—and was able to continue using the same application without modification. He then added two sensors to detect when doors were opened so that he could be notified when unexpected activity occurred. This used our door-window monitoring application sends email notifications, which can contain images from any cameras in the home. The user later added two light controllers and another application to control them. What started off as simply wanting to see his front yard from work evolved into a notification system and lighting control.

Diagnostic support in interoperability protocols On the negative side, at least two homes had problems diagnosing their deployments. For instance, when applications that use Z-Wave devices behaved unexpectedly, users could not easily tell if it was due to code bugs, device malfunctions or poor signal strength to the device. Disambiguation requires effort and technical expertise (e.g., unmount the device, bring it close to the controller, and then observe application behavior).

This difficulty is an instance where the added complexity of network devices, in contrast to directly connected peripherals, becomes apparent. Countering it requires diagnostic tools but they are hard to build today because interoperability protocols have limited diagnostic support. We thus recommend that device protocols be extended to provide diagnostic information. Even something akin to ICMP would be a step forward.

6 Evaluation

In addition to our experience with real homes and developers, we evaluate HomeOS through controlled experiments, focusing on its ease of programming, ease of managing and system performance. This gives us quantitative validation to confirm our real-world experiences above. We find that developers can write realistic applications within 2 hours, that users can use our management interfaces with similar success to other carefully designed systems and that system performance is good enough to easily support rich, interactive applications.

6.1 Ease of programming

To evaluate how easy it is to write a HomeOS application, we conducted a study where we recruited student and researcher volunteers to develop HomeOS applications. (Different from the students mentioned in §5.)

We provided our participants with a brief introduction to HomeOS, some basic documentation on our abstractions, all the drivers, and four simple applications that use only one driver each—image recognition, camera snapshot, DLNA music player, and light-switch controller. Each participant got a total of five-minutes of verbal instructions (with no demonstration of code) on the goal of the study and pointers to these resources. We left the participant and the testbed, with the HomeOS server console running an IDE (Visual Studio) configured to use HomeOS binaries. We provided little assistance beyond the initial training, though on three occasions the participants uncovered bugs in our system that we had to fix before they could proceed.

We gave each participant the task of writing one of two applications for our testbed. "Custom Lights Per-User" (CLU) will adjust the lights in any room based on its occupant's preferences. This application needs to find cameras in the house to which it has access, continuously poll them, use the image recognition service to identify the occupant (if any), and set the "dimmers" in the camera's room to the occupant's preferences. For testing, we gave each participant two photographs on which the image recognition service was trained and the user-to-lighting preference chart.

The second application—"Music Follows the Lights" (MFL)—was one we previously built but did not provide to participants. This application finds all lights and media devices in the house, registers for changes to the lights' status and plays music (from a media server) on an audio device in rooms with lights that are on.

We recruited ten participants for this study via a mailing list within our organization. Seven were graduate students and three were researchers. Only one had prior experience with home automation, and none had significant prior experience with programming service-based abstractions (e.g., WSDL or SOAP). This level of expertise is at the low-end of what we expect of future HomeOS developers. We gave each participant two hours to write an application. Half of the participants were given the first application and half were given the second.

Figure 8(a) summarizes the results of our study. The time reported was computed from the end of verbal instructions until the participant was done, minus any breaks the participant took and the time we spent correcting bugs. Eight participants developed complete applications within approximately two hours (126 minutes). Of the two who did not finish, one spent the bulk of the time developing a "slick GUI" for the application instead of its core logic and the other did not realize that HomeOS drivers were not running by default. This problem stemmed from a misinterpretation of instructions and could have been avoided with clearer instructions.

In the exit interview, almost all participants (even those who did not finish) reported that HomeOS was "very programmable" and the APIs were "natural." However, they also expected more syntactic support in the IDE for invoking operations. We are addressing this issue by defining a C# interface for each role.

These results suggest that it is easy to develop applications for HomeOS. Even without prior experience, developers were able to implement realistic applications in just a couple of hours. We do not mean to suggest that all HomeOS applications can be developed in two hours. Our study emphasized the use of HomeOS's basic abstractions and did not require the developers to focus on consumer-facing issues such as a richer GUI. However, it does provide evidence that the base programming abstractions are a good fit for applications in the home.

6.2 Ease of managing

Our second study evaluates whether our management primitives and interfaces are easy enough for non-expert home users to use. We find that with no training, typical home users are able to complete typical management tasks correctly around 80% of the time.

Methodology We began each session by explaining the background and goals of the study and the three security principals—users, devices and applications. We asked participants to pretend to be a member of the following imaginary family. Jeff and Amy are husband and wife. Dave and Rob are their eight-year and seven-month old kids. Jeff's brother Sam, who visits occasionally, has a guest account. The house has 29 devices of nine different types. Three of the devices—camera and microphone in Rob's room, and the front door lock—are high-security. The family has four applications for lighting, monitoring, and temperature control, and fourteen rules specifying access controls policies. We assigned the male participants to play Jeff and females to play Amy.

We then asked them to perform the 7 management tasks show in Figure 8(b) using our UI. These tasks reflect what we expect users to do with HomeOS and span key management tasks (§4.4). Tasks 1, 2 and 6 require configuring applications, including restricting their use to certain users, devices and times of day. Task 3 requires configuring a new device with group and application access. Task 4 requires adding a new guest. Tasks 5 and 7 require verifying policies based on specific concerns.

At the same time, our tasks stress the ability of primitives in HomeOS to simplify management. For instance, Tasks 1 and 2 use application manifests, Datalog rules,

	app	LoC	mins
1	CLU	183	84
2	CLU	193	62
3	CLU	156	66
4	CLU	172	113
5	CLU	221	107
6	MFL	224	95
7	MFL	244	126
8	MFL	239	102
9	MFL	303	93*
10	MFL	130	100*

(a) Programming study

Task	✓
1. Configure your new EcoMonitor app. Let it access all but high-security devices for everyone.	11
2. Configure your new MusicFollowsMe app. Let it access all motion detectors and speakers but no high-security devices. All residents can use it but kids cannot not play music in the parents' bedroom	9
3. Configure your new kitchen security camera. Mark it high-security and let HomeMonitor access it.	11
4. Give guest access to Jane, who will be visiting until September 6th. Place her in the Guests group so that she can use appropriate apps during her visit.	12
5. Check the rules and tell the facilitator which apps can access high security devices.	1
6. Configure your new OpenFrontDoor app. Residents can use it any time. Sam (guest) cannot use it at all. Jane (guest) can use it only during the day (8 AM to 8 PM).	11
7. Check if only adults can access the camera in Rob's room and only using HomeMonitor.	10

(b) Management study

Figure 8: Results of our two studies: (a) The application developed and time taken by each participant, '*' implies an incomplete program; (b) Assigned management tasks and the number of participants (of 12) that completed each accurately

and user and device groups, whereas Task 4 uses time-based user accounts and user groups. Ideally, we would evaluate each primitive separately but we found management tasks that stress only one primitive to be unrealistic.

We simulated a real-world setting by not training our participants in using the technology they are required to manage. Instead we provided manuals for each task type and told them that reading the manuals was not required but they could refer to them anytime if they wanted.

Participants We had twelve participants (eight male, four female) from the Greater Puget Sound region in Washington state. We recruited the participants through a professional recruiting service. We screened them to ensure they are somewhat familiar with home technology. They were required to own at least one TV, one computer, and three types of electronic devices (e.g., wireless router, security camera, smartphone, etc.). They were also required to be able to conduct basic administrative tasks (e.g., set up an account on a computer)[2].

Results Figure 8(b) shows the number of participants that completed each task correctly. We see that the accuracy rate is high. Overall, it is 77%; ignoring Task 5 (discussed below), it is 89%. This result is encouraging because it was obtained without any training and many participants did not use the manual. For reference, we note that our accuracy rate is similar to that obtained with careful design of system semantics and interfaces for file system access control [38] and firewall configuration [37]. Our participants took one to four minutes to complete individual tasks.

While the errors in most tasks were simply forgetting a single click, Task 5 was particularly problematic. Only one participant was able to complete it correctly. Others had difficulty forming the correct query for the task using our faceted browsing interface. (Forming the query for Task 7, which also used the same interface, was not as problematic.) They could correctly and easily tell that the HomeMonitor application was using the camera, but did not realize that so was another application. We plan to address this by augmenting the UI to reduce the work needed to detect things that are unexpected or the absence of use by other applications. We believe that the underlying primitives do not need modification.

In the exit interview, we asked the participants how easy HomeOS was to use and learn, on a 7-pt. Likert scale from "Strongly Disagree" (1) to "Strongly Agree" (7). The participants found the system easy to learn (avg. 6.0), easy to use (avg. 6.0), and intuitive (avg. 5.5).

6.3 System performance

In addition to easy programming and management, HomeOS must have acceptable performance. Our goals are to have latency that is low enough to run responsive, interactive applications and to offer scalability and throughput that can handle large, complex homes. To quantify the overhead of layering in HomeOS, we compare against a hypothetical, monolithic system without layering and isolation.

Experimental setup To gather performance data about HomeOS, we ran a simple benchmarking application using a virtual device on a quad-core Intel Xeon 2.67 GHz PC. Unless otherwise specified, there is one application and one driver running. The application generates load by creating ten continuous tasks that attempt to invoke an operation on the device at a fixed rate, but are scheduled by .NET ThreadPool which dynamically picks a number of threads to execute based on current performance.

Latency of operation invocation Figure 9 shows the latency of an operation invocation with no arguments under different loads. While we incur higher overhead than without isolation, the difference is approximately only 25% or a few hundred microseconds. Even under heavy load, we are able to keep latencies below 2 ms. This is two orders of magnitude lower than the interactive response time guideline of 100 ms [18], which means that

[2]While this may seem to exclude typical home users, homes typically have at least one experienced tech guru [36].

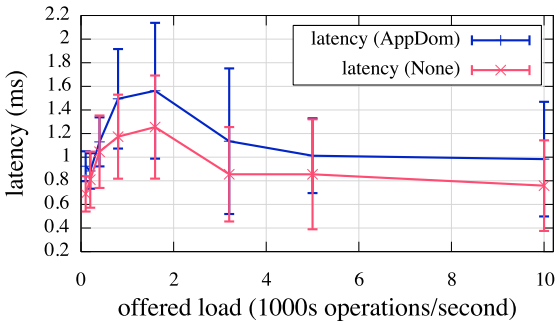

Figure 9: Operation invocation latency as a function of offered load with and without Application Domain isolation

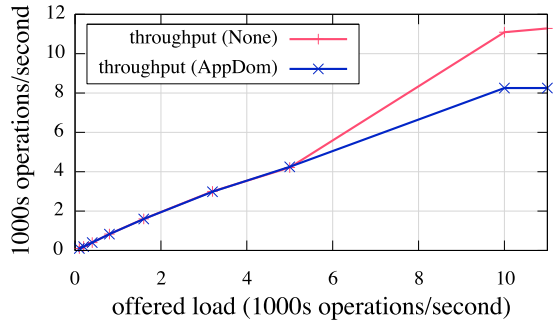

Figure 10: Operation invocation throughput as a function of offered load with and without Application Domain isolation

applications can compose several services and allow for network delays before responding to the user.

The odd increase in latency when the offered load is low (between 100 and 3000 operations per second, respectively) is an artifact of the ThreadPool scheduling rapidly switching between threads when each thread offers substantially lower load than what the machine can handle. We found it to persist across other microbenchmarks as well.

Throughput of operations To evaluate the load that HomeOS can handle, we tracked the throughput of the system at different offered loads both with and without AppDomain isolation. Figure 10 shows that the throughput of the two modes mirror each other until the system is driven near peak throughput. With AppDomain isolation, HomeOS handles approximately 8,250 operation invocations per second, while with no isolation the system can handle nearly 11,300 operation invocations per second. Beyond that load, ThreadPool scheduling backs off. If an application does not use this or a similar mechanism, latencies climb substantially when the system is driven past the load it can handle as one would expect. This level of performance has been well-beyond what was required for any of our current deployments.

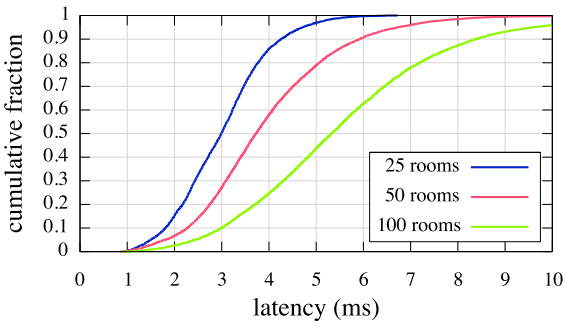

Figure 11: Cumulative fraction of operation invocations completed within a given latency for varying numbers of applications and devices

Scalability To understand how HomeOS scales to a large home with many devices and applications, we examine the latency of operation invocation in an extreme setting. We emulate a large home with a varying number of rooms, each room containing 4 devices and one highly-active application querying each device 10 times per second. Figure 11 shows the operation invocation latency in this setting. As we increased the number of rooms from 25 to 50 to 100 (1,000, 2,000 and 4,000 total operations per second), we see a median latency of 3–5.5 ms depending on the number of applications. The latency in this experiment is higher than those presented earlier at the same total load because of the added overhead of having 125–500 modules running rather than two. The threads across modules are not managed as part of the same ThreadPool. Despite this additional overhead, the vast majority of operations complete within 10 ms, which is an order of magnitude below the interactive guideline of 100 ms. This result suggests that HomeOS can easily scale to large, well-connected homes.

7 Related work

While we draw on many strands of existing work, we are unaware of a system similar to HomeOS that provides a PC abstraction for home technology to simplify management and application development while remaining extensible. We categorize related work into five groups.

1. Device interoperability Many systems and standards for providing device-to-device interoperability in the home exist. They include DLNA [14], UPnP [41], Z-Wave [46], ZigBee [45], and Speakeasy [16]. HomeOS is agnostic to what interoperability standards are used and can incorporate any of these. But, as discussed earlier, while interoperability is helpful, it does not provide enough support for users and developers.

2. Multi-device systems Many commercial home automation and security systems integrate multiple devices in the homes. Like HomeOS, they centralize control, but such systems tend to be monolithic and hard for users to extend. For instance, Control4 [11]—one of the most extensible automation systems—allows for only its own devices and a limited set of ZigBee devices; and offers only a limited form of programming based on rules of the form "upon Event E, do Task T." Further, the technical complexity of installing and configuring Control4 and other systems (e.g., HomeSeer [28], Elk M1 [17] and Leviton [30]) can be handled only by professional installers (which is expensive) or expert hobbyists. In the research community, EasyLiving [29] was a monolithic system with a fixed set of applications integrated into the platform. In contrast to such systems, we focus on building a system that can be extended easily with new devices and applications by non-experts.

3. Programmability for the home We proposed the idea of a home-wide OS in a position paper [13]. This paper presents an architecture based on our experiences, a more complete system, and its evaluation.

Newman proposes using "recipes," [34] which are programs in a domain-specific language that compose devices in the home. He also advocates using marketplaces to disseminate recipes. With HomeOS applications, our vision is similar. Newman does not discuss how recipes can be realized in practice, which requires tackling challenges similar to those we address.

Previous work also advocates using a central controller to simplify integration [20, 39]. They have a different scope than HomeOS. For instance, Rosen et al. [39] (incidentally, also called HomeOS), focus on providing context such as user location to applications. These works offer little detail about their design and implementation.

Other systems have employed services in the home environment. iCrafter is a system for UI programmability [35]. ubiHome aims to program ubiquitous computing devices inside the home using Web services [24]. While our use of the service abstraction is similar to these systems, we engineer a more complete system for programming and managing devices in the home.

4. Management challenges in the home Calvert et al. outline the various management challenges in the home network [8], and like us, argue for centralization. We go beyond management issues and also focus on simplifying application development. Further, in contrast to their proposal, we do not require device modifications.

5. OSes for network devices Researchers have designed OSes over multiple devices in other domains. iROS aims to simplify programming devices such as displays and white-boards in collaborative workspaces [6]. NOX aims to simplify managing switches in enterprise networks [23]. While conceptually similar, HomeOS handles complexities specific to the home environment.

8 Conclusions and future work

HomeOS simplifies the task of managing and extending technology in the home by providing a PC-like abstraction for network devices to users and developers. Its design is based on management primitives that map to how users want to manage their homes, protocol-independent services that provide simple APIs to applications and a kernel that is agnostic of the functionality and protocols of specific devices. Experience with real users and developers, in addition to controlled experiments, help validate the usefulness of the abstraction and our design.

This experience also reveals gaps where we could not cleanly implement the abstraction due to limitations of device protocols (e.g., little support for diagnosis and incompatible implementations across vendors) or due to limited features being exposed by devices over the network. We plan to address these limitations in the future.

More broadly, our hope is that this work spurs the research community to further explore the home as a future computing platform. While we cannot outline a complete agenda for work in this area, we point out two fruitful directions based on our experience:

1. Foundational services Over the years PC applications have come to expect some essential services that the OS provides (e.g., a file system). Are there similar services for the home environment? Such services should not only be broadly useful but also almost universally implementable. For instance, consider occupancy information—which rooms are currently occupied by people. It can benefit many applications (e.g., lighting control, thermostat control, and security), but depending on the devices in the home, it may be difficult to infer reliably (e.g., motion sensors can be triggered by pets; cameras are more reliable). Making occupancy an essential service requires each home to possess the necessary devices, thus increasing the cost of a basic HomeOS installation. (This is akin to PC or smartphone OSes specifying minimum hardware requirements.) Thus, careful consideration is needed to determine which services a system like HomeOS should provide in all homes.

2. Identity inference Some desired reactions to physical actions in the home depend on the identity of the user or who else is around. For instance, users may want to play different music based on who entered and turned

on the lights, or parents may not want their children to turn on the Xbox in their absence. Currently, HomeOS can either not support such policies (lightswitches have no interface to query user identity) or support them in an inconvenient manner (ask parents for their password). A promising avenue for future work is to build non-intrusive identity inference (e.g., using cameras in the home, or users' smartphones), and then allow users to express policies based on that inference. A key challenge in realizing this system is to maintain safety in the face of possible errors in identify inference.

Acknowledgements Tom Anderson, S. Keshav, Ed Nightingale, Andrew Warfield and our anonymous reviewers provided helpful comments on this paper. The volunteers who were brave enough to run HomeOS in their homes, the students who developed HomeOS applications and drivers, and our study participants provided valuable feedback that helped refine our design. Frank Martinez was our first volunteer and helped set up the volunteer program. We are grateful to them all.

References

[1] Home security systems, home security products, home alarm systems - ADT. http://www.adt.com.

[2] Application Domains. http://msdn.microsoft.com/en-us/library/2bh4z9hs%28v=vs.100%29.aspx.

[3] O. Ardakanian, S. Keshav, and C. Rosenberg. Markovian Models for Home Electricity Consumption. In *SIGCOMM Workshop on Green Networking*, 2011.

[4] N. Banerjee, S. Rollins, and K. Moran. Automating Energy Management in Green Homes. In *SIGCOMM Workshop on Home Networks (HomeNets)*, 2011.

[5] L. Bauer, L. Cranor, R. W. Reeder, M. K. Reiter, and K. Vaniea. A user study of policy creation in a flexible access-control system. In *CHI*, 2008.

[6] J. Borchers, M. Ringel, J. Tyler, and A. Fox. Stanford Interactive Workspaces: A Framework for Physical and Graphical User Interface Prototyping. *IEEE Wireless Communications. Special Issue on Smart Homes*, 2002.

[7] A. J. Brush, B. Lee, R. Mahajan, S. Agarwal, S. Saroiu, and C. Dixon. Home Automation in the Wild: Challenges and Opportunities. In *CHI*, 2011.

[8] K. L. Calvert, W. K. Edwards, and R. E. Grinter. Moving Toward the Middle: The Case Against the End-to-End Argument in home networking. In *HotNets*, 2007.

[9] S. Ceri, G. Gottlob, and L. Tanca. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, 1, 1989.

[10] A. Chaudhuri, P. Naldurg, G. Ramalingam, S. Rajamani, and L. Velaga. EON: Modeling and Analyzing Access Control Systems with Logic Programs. In *CCS*, 2008.

[11] Control4 Home Automation and Control. http://www.control4.com.

[12] Crestron Electronic: Home automation, building and campus control. http://www.crestron.com.

[13] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and V. Bahl. The home needs an operating system (and an app store). In *HotNets*, 2010.

[14] DLNA. http://www.dlna.org/home.

[15] W. K. Edwards, R. E. Grinter, R. Mahajan, and D. Wetherall. Advancing the state of home networking. *Communications of the ACM*, 54, 2011.

[16] W. K. Edwards, M. W. Newman, J. Z. Sedivy, T. F. Smith, D. Balfanz, D. K. Smetters, H. C. Wong, and S. Izadi. Using SpeakEasy for ad hoc peer-to-peer collaboration. In *CSCW*, 2002.

[17] M1 Security & Automation Controls. http://www.elkproducts.com/m1_controls.html.

[18] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer. Using latency to evaluate interactive system performance. In *OSDI*, 1996.

[19] ENVI whole home energy monitor by powersave. http://www.currentcost.net.

[20] C. Escoffier, J. Bourcier, P. Lalanda, and J. Yu. Towards a home application server. In *Consumer Communication & Networking Conference*, 2008.

[21] FFmpeg. http://ffmpeg.org.

[22] R. E. Grinter, W. K. Edwards, M. Chetty, E. S. Poole, J.-Y. Sung, J. Yang, A. Crabtree, P. Tolmie, T. Rodden, C. Greenhalgh, and S. Benford. The ins and outs of home networking: The case for useful and usable domestic networking. *ToCHI*, 16(2), 2009.

[23] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. *SIGCOMM CCR*, 38(3), 2008.

[24] Y.-G. Ha, J.-C. Sohn, and Y.-J. Cho. ubiHome: An Infrastructure for Ubiquitous Home Network Services. In *IEEE International Symposium on Consumer Electronics*, 2007.

[25] M. Hearst, A. Elliott, J. English, R. Sinha, K. Swearingen, and K.-P. Yee. Finding the flow in web site search. *Communications of the ACM*, 45(9), 2002.

[26] HomeOS. http://homeos.codeplex.com.

[27] HomeOS demos. http://research.microsoft.com/en-us/um/redmond/projects/homeos/homeos-demos.htm.

[28] Home Automation Systems - HomeSeer. http://www.homeseer.com.

[29] J. Krumm, S. Harris, B. Meyers, B. Brumitt, M. Hale, and S. Shafer. Multi-Camera Multi-Person Tracking for EasyLiving. In *IEEE Workshop on Visual Surveillance*, 2000.

[30] Leviton Online Store - LevitonProducts.com. http://www.levitonproducts.com.

[31] H. M. Levy. *Capability Based Computer Systems*. Digital Press, 1984.

[32] N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *International Symposium on Practical Aspects of Declarative Languages*, 2003.

[33] M. L. Mazurek, J. Arsenault, J. Breese, N. Gupta, I. Ion, C. Johns, D. Lee, Y. Liang, J. Olsen, B. Salmon, R. Shay, K. Vaniea, L. Bauer, L. F. Cranor, G. R. Ganger, and M. K. Reiter. Access Control for Home Data Sharing: Attitudes, Needs and Practices. In *CHI*, 2010.

[34] M. W. Newman. Now we're cooking: Recipes for end-user service composition in the digital home. In *IT@Home: Workshop associated with CHI*, 2006.

[35] S. R. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd. ICrafter: A Service Framework for Ubiquitous Computing Environments. In *Ubicomp*, 2001.

[36] E. S. Poole, M. Chetty, R. E. Grinter, and W. K. Edwards. More than Meets the Eye: Transforming the User Experience of Home Network Management. *Designing Interactive Systems*, 2008.

[37] F. Raja, K. Hawkey, and K. Beznosov. Revealing hidden context: improving mental models of personal firewall users. In *Symposium on Usable Privacy and Security (SOUPS)*, 2009.

[38] R. W. Reeder, L. Bauer, L. F. Cranor, M. K. Reiter, and K. Vaniea. More than skin deep: measuring effects of the underlying model on access-control system usability. In *CHI*, 2011.

[39] N. Rosen, R. Sattar, R. W. Linderman, R. Simha, and B. Narahari. HomeOS: Context-Aware Home Connectivity. In *International Conference on Pervasive Computing and Applications*, 2004.

[40] Security Policy Assertion Language implementation for Microsoft .NET. http://research.microsoft.com/secpal.

[41] Universal Plug-and-Play. http://www.upnp.org.

[42] S. VanDeBogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazieres. Labels and Event Processes in the Asbestos Operating System. *TOCS*, 25(4), 2007.

[43] J. Wu, A. Osuntogun, T. Choudhury, M. Philipose, and J. M. Rehg. A Scalable Approach to Activity Recognition based on Object Use. In *International Conference on Computer Vision*, 2007.

[44] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres. Securing distributed systems with information flow control. In *NSDI*, 2008.

[45] ZigBee Alliance. http://www.zigbee.org.

[46] Z-Wave.com - ZwaveStart. http://www.z-wave.com.