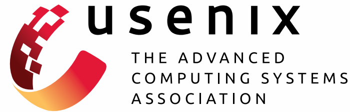

Beam: Ending Monolithic Applications for Connected Devices

Chenguang Shen, *University of California, Los Angeles;* Rayman Preet Singh, *Samsung Research;* Amar Phanishayee, Aman Kansal, and Ratul Mahajan, *Microsoft Research*

https://www.usenix.org/conference/atc16/technical-sessions/presentation/shen

This paper is included in the Proceedings of the
2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Open access to the Proceedings of the
2016 USENIX Annual Technical Conference
(USENIX ATC '16) is sponsored by USENIX.

Beam: Ending Monolithic Applications for Connected Devices

Chenguang Shen (UCLA)*   *Rayman Preet Singh* (Samsung Research)*
Amar Phanishayee   *Aman Kansal*   *Ratul Mahajan*
Microsoft Research

Abstract— The proliferation of connected sensing devices (or *Internet of Things*) can in theory enable a range of applications that make rich inferences about users and their environment. But in practice developing such applications today is arduous because they must implement all data sensing and inference logic, even as devices move or are temporarily disconnected. We develop Beam, a framework that simplifies IoT applications by letting them specify "what should be sensed or inferred," without worrying about "how it is sensed or inferred." Beam introduces the key abstraction of an *inference graph* to decouple applications from the mechanics of sensing and drawing inferences. The inference graph allows Beam to address three important challenges: (1) device selection in heterogeneous environments, (2) efficient resource usage, and (3) handling device disconnections. Using Beam we develop two diverse applications that use several different types of devices and show that their implementations required up to 12× fewer source lines of code while resulting in up to 3× higher inference accuracy.

1 Introduction

Connected sensing devices, such as cameras, thermostats, in-home motion, door-window, energy, water sensors [2], collectively dubbed as the *Internet of Things* (IoT), are rapidly permeating our living environments [3], with an estimated 50 billion such devices in use by 2020 [34]. In theory, they enable a wide variety of applications spanning security, efficiency, healthcare, and others. But in practice, developing IoT applications is arduous because the tight coupling of applications to specific hardware requires each application to implement the data collection logic from these devices and the logic to draw inferences about the environment or the user.

Unfortunately, this monolithic approach where applications are tightly coupled to the hardware, is limiting in two important ways. First, for application developers, this complicates the development process, and hinders broad distribution of their applications because the cost of deploying their specific hardware limits user adoption. Second, for end users, each sensing device they install is limited to a small set of applications, even though the hardware capabilities may be useful for a broader set of applications. How do we break free from this monolithic and restrictive setting? Can we enable applications to be programmed to work seamlessly in heterogeneous environments with different types of connected sensors and devices, while leveraging devices that may only be available opportunistically, such as smartphones and tablets?

To address this problem, we start from an insight that many inferences required by applications can be drawn using multiple types of connected devices. For instance, home occupancy can be inferred by either detecting motion or recognizing people in images, with data sampled from motion sensors (such as those in security systems or Nest [12]), cameras (e.g. Dropcam [4], Simplicam [18]), microphone, smartphone GPS, or using a combination of these sensors, since each may have different sources of errors. *We posit that inference logic, traditionally left up to applications, ought to be abstracted out as a system service, thus decoupling "what is sensed and inferred" from "how it is sensed and inferred"*. Such decoupling enables applications to work in heterogeneous environments with different sensing devices while at the same time benefiting from shared and well trained inferences. Consequently, there are three key challenges in designing such a service:

Device selection: The service must be able to select the appropriate devices in a deployment that can satisfy an application's inference request (including inference accuracy). Device selection helps applications to run in heterogeneous deployments. It also helps applications to operate in settings with user mobility where the set of usable devices may change over time. Moreover, applications can leverage multiple available devices to improve inference accuracy, as shown in Figure 1.

Efficiency: For inferences that are computationally ex-

*Work done during an internship at Microsoft Research

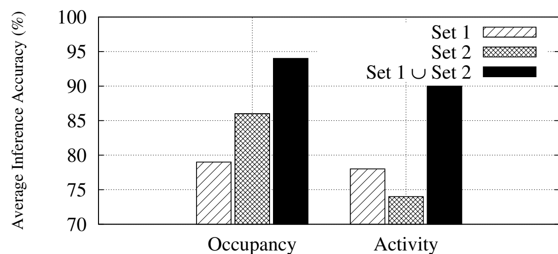

Figure 1: Improvement in occupancy and activity inference accuracy by combining multiple devices in a lab deployment. For occupancy, sensor set 1 = {camera, microphone} in one room and set 2 ={PC interactivity detection} in a second room. For physical activity, set 1 = {phone accelerometer} and set 2 = {wrist worn FitBit [5]}.

pensive to run locally on user devices, or to support deployments that span geographical boundaries, the service should be able to offload computation to remote servers. In doing so, the service should partition computation while efficiently using network bandwidth.

Disconnection tolerance: The service should be able to handle dynamics that can arise due to device disconnections and user mobility.

To address these challenges concretely, we propose *Beam*, an application framework and associated runtime which provides applications with inference-based programming abstractions. It introduces the key abstraction of an **inference graph** to not only decouple applications from the mechanics of sensing and drawing inferences, but also directly aid in addressing the challenges identified above. Applications simply specify their inference requirements, while the Beam runtime bears the onus of identifying the required sensors in the given deployment and constructing an appropriate inference graph.

Inference graphs are made up of modules which are processing units that encapsulate inference algorithms; modules can use the output of other modules for their processing logic. Beam introduces three simple building blocks that are key to constructing and maintaining the inference graph: typed inference data units (IDUs) which guide module composability, channels that abstract all inter-module communications, and coverage tags that aid in device selection. The Beam runtime instantiates the inference graph by selecting suitable devices and assigning computational hosts for each module. Beam also mutates this assignment by partitioning the graph at runtime for efficient resource usage. Beam's abstractions and runtime together provide disconnection tolerance.

Our implementation of the Beam runtime works across Windows PCs, tablets, and phones. Using the framework, we develop two realistic applications, eight different types of inference modules, and add native support for many different types of sensors. Further, Beam supports all device abstractions provided by HomeOS [33], thus enabling the development of a variety of inference modules. We find that for these applications: 1) using Beam's abstractions results in up to 4.5× fewer development tasks and 12× fewer source lines of code with negligible runtime overhead; 2) inference accuracy is 3× higher due to Beam's ability to select devices in the presence of user mobility; and 3) network resource usage due to Beam's dynamic graph partitioning matches hand-optimized versions for the applications.

2 Beam Overview

In this section, we first describe two representative classes of applications and distill the challenges an inference framework should address. Next, we describe the key abstractions central to Beam's design in addressing the identified challenges.

2.1 Example Applications

Our motivation for designing Beam are data-driven-inference based applications, aimed at homes [12, 19], individual users [11, 14, 59, 69, 72] and enterprises [8, 16, 24, 46, 60]. We identify the challenges of building an inference framework by analyzing two popular application classes in detail, one that infers environmental attributes and another that senses an individual user.

Rules: A large class of popular applications is based on the 'If This Then That (IFTTT)' pattern [9, 67]. IFTTT enables users to create their own rules connecting sensed attributes to desired actions. We consider a particular rules application which alerts a user if a high risk appliance, e.g., electric oven, is left on when the home is unoccupied [64]. This application uses the appliance-state and home occupancy inferences.

Quantified Self (QS) [11, 14, 23, 35, 53] disaggregates a user's daily routine by tracking her physical activity (walking, running, etc), social interactions (loneliness), mood (bored, focused), computer use, and more.

Using these two popular classes of applications we address three important challenges they pose: device selection, efficiency, and disconnection tolerance, as detailed in Section 1. Next, we explain the key abstractions in Beam aimed at tackling these challenges.

2.2 Beam Abstractions

In Beam, *application developers* only specify their desired inferences. To satisfy the request, Beam bears the onus of identifying the required sensors and inference algorithms in the given deployment and constructing an inference graph.

Inference Graphs are directed acyclic graphs that connect devices to applications. The nodes in this graph correspond to *inference modules* and edges correspond to *channels* that facilitate the transmission of *inference data units (IDUs)* between modules. While these abstractions

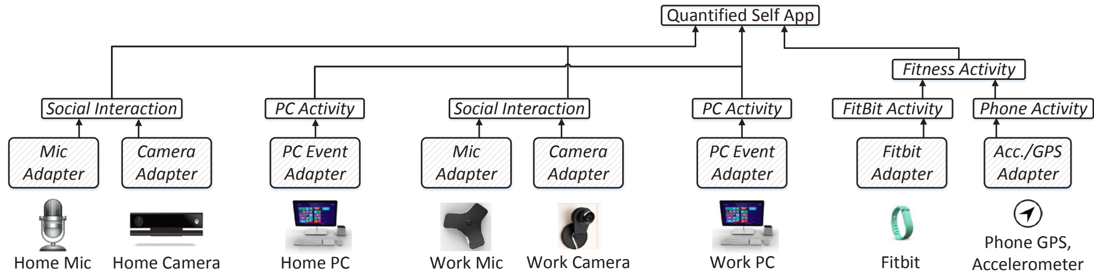

Figure 2: Inference graph of modules for the Quantified Self (QS) app. Adapters are device driver modules.

are described in more detail below, Figure 2 shows an example inference graph for the QS application that we later build and evaluate. The graph uses eight different devices spread across the user's home and workplace, and includes mobile and wearable devices. The application requests a top-level inference as an IDU and Beam dynamically selects the modules that can satisfy this inference based on the devices available. For example, in Figure 2, to satisfy the application's request for inferences pertaining to fitness activities Beam uses a module that combines inferences drawn separately from a user's smartphone GPS, accelerometer, and Fitbit device, thus forming part of the inference graph for QS. Figure 3 shows the inference graph for the Rules application.

Composing an inference as a directed graph enables sharing of data processing modules across applications and other modules that require the same input. In Beam, each computing device associated with a user, such as a tablet, phone, PC, or home hub, has a part of the runtime, called the **Engine**. Engines are computational hosts for inference graphs. Figure 4 shows two engines, one on the user's home hub and another on her phone; the inference graph for QS (shown in Figure 2) is split across these engines, while the QS application runs on a cloud server. For simplicity, we do not show another engine that may run on the user's work PC.

IDU: An *Inference data unit (IDU)* is a typed inference, and in its general form is a tuple $<t,e,s>$, which denotes any inference with state information s, generated by an inference algorithm at time t and error e. The types of the inference state s, and error e, are specific to the inference at hand. For instance, s may be of a numerical type such as a double (e.g., inferred energy consumption), or an enumerated type such as high, medium, or low. Similarly, error e may specify a confidence measure (e.g., standard deviation), probability distribution, or error margin (e.g., radius). IDUs abstract away "what is inferred" from "how it is inferred". The latter is handled by inference modules, which we describe next.

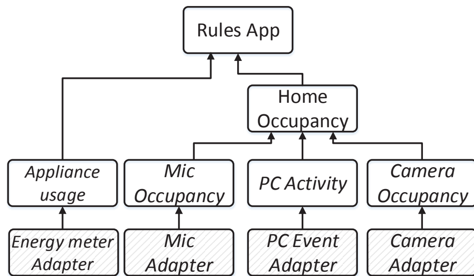

Figure 3: Inference graph for the Rules application.

Inference Modules: Beam encapsulates inference algorithms into modules. Inference modules consume IDUs from one or more modules, perform certain computation using IDU data and pertinent in-memory state, and output IDUs. Special modules called *adapters* interface with underlying sensors and output sensor data as IDUs. Adapters are device drivers that decouple "what is sensed" from "how it is sensed". *Inference developers* specify (i) a module's input dependencies (either as IDU types or as modules), (ii) the IDU type it generates, and (iii) its configuration parameters. Modules have complete autonomy over how and when to output an IDU, and can maintain arbitrary internal states. Listing 1 shows a specification for the Home Occupancy inference module in the Rules inference graph (Figure 3). It lists (i) input dependencies of PC Activity *OR* Mic Occupancy *OR* Camera Occupancy, (ii) *HomeOccupancyIDU* to be the type of output it generates, and (iii) a control parameter, *sampleSize*, that specifies the temporal size of input samples (in seconds) to consider in the inference logic. Application developers request the local engine for desired infer-

```
1 <Spec>
2   <ControlParameters> <!-- Module parameters -->
3     <Param name="sampleSize" type="int" value="5"/>
4   </ControlParameters>
5
6   <Output> <!-- Output channel IDU spec -->
7     <Inference type="Beam.IDU.HomeOccupancyIDU"/>
8   </Output>
9
10  <Input> <!-- Input channels -->
11    <InputBlock type="OR">
12      <InputChannel Mode="FreshPush">
13        <Module type="Beam.Modules.PCActivity"/>
14        <Module type="Beam.Modules.MicOccupancy"/>
15        <Module type="Beam.Modules.CameraOccupancy"/>
16      </InputChannel>
17    </InputBlock>
18  </Input>
19 </Spec>
```

Listing 1: Module specification of Home Occupancy.

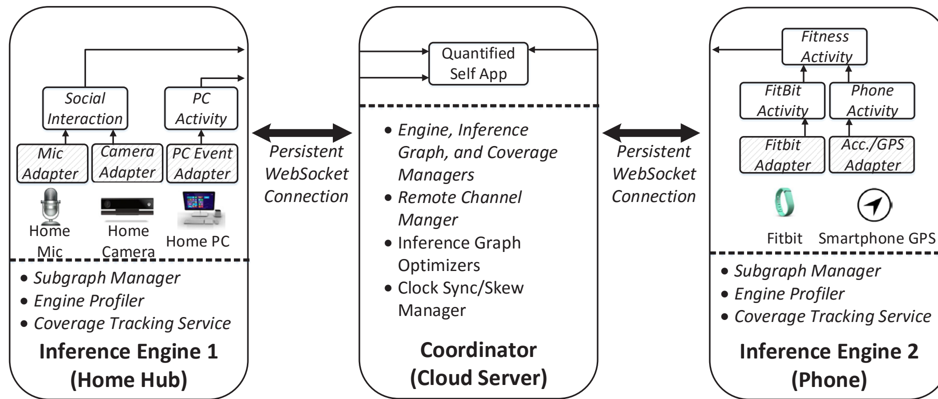

Figure 4: An overview of different components in an example Beam deployment with 2 Engines.

ences, for example:

```
engineInstance.Request(
    Beam.Modules.ModHomeOccupancy,
    tags, Mode.FreshPush);
```

These are satisfied by inference modules implemented by inference developers, and applications receive IDUs via a callback.

Channels: To ease inference composition, *channels* link modules to each other and to applications, abstracting away the complexities of connecting modules across different devices. Channels provide support for disconnections tolerance and enable optimizations such as batching IDU transfers for efficiency. Every channel has a single *writer* and a single *reader* module. Modules can have multiple input and output channels. Channels connecting modules on the same engine are *local*. Channels connecting modules on two different engines, across a local or wide area network, are *remote* channels. Remote channels enable applications and inference modules to seamlessly use remote devices or modules. Channels can be either configured to deliver IDUs to the reader as soon as the writer pushes it (*FreshPush*, as seen in Listing 1 line 12), or to deliver IDUs in batches thus amortizing the cost of computation and network transfers.

Coverage Tags: Coverage tags help manage sensor coverage. Each adapter is associated with a set of coverage tags which describes what the sensor is sensing. For example, a location string tag can indicate a coverage area such as “home” and a remote monitoring application can use this tag to request an occupancy inference for this coverage area. Coverage tags are strongly typed. Beam uses tag types only to differentiate tags and does not dictate tag semantics. This gives applications complete flexibility in defining new tag types. Adapters are assigned tags by the respective engines at setup time, and are updated at runtime to handle dynamics (Section 3.1).

Beam’s runtime also consists of a **Coordinator** which interfaces with all engines in a deployment and runs on a replicated server that is reachable from all engines. The coordinator maintains remote channel buffers to support reader or writer disconnections (typical for mobile devices). It also provides a place to reliably store state of inference graphs at runtime while being resistant to engine crashes and disconnections. The coordinator is also used to maintain reference time across all engines. Engines interface with the coordinator using a persistent web-socket connection, and instantiate and manage the parts of inference graphs local to them.

3 Beam Runtime

In this section, we describe how the Beam runtime uses the inference graph to aid in device selection, efficient graph partitioning, and handling device disconnections.

3.1 Device Selection

Beam simplifies application development by automatically selecting devices that match its inference request in heterogeneous deployments and in the presence of user mobility. Beam leverages the device discovery mechanism in HomesOS [33] to discover and instantiate adapter modules for available sensors in the deployment.

Applications request their local Beam engines for all inferences they require, including the coverage associated with each inference. All application requests are forwarded to the coordinator. Using inference module specifications and devices with matching coverage tags available in the deployment [1], the coordinator recursively resolves all required inputs of each module. A module’s coverage tag set includes tags from the downstream modules it processes data from.

Handling environmental dynamics: Movement of users and devices can change the set of sensors and devices that satisfy an application’s requirement. For instance, consider an application that requires camera input from the device currently facing the user at any time, such as the camera on her home PC, work PC, or smartphone. In such scenarios, the inference graph needs to

[1] The requested tag must match one of the adapter tags.

be updated dynamically. Beam updates the coverage tags to handle such dynamics. Tags of *location* type (e.g., "home") are assumed to be static and are only edited by the user. For tags of type *user*, the sensed subject is mobile and hence the sensors that cover it may change. The coordinator's *tracking service* manages the coverage tags associated with adapters on various engines.

The user tracking service updates the coverage tags as the user moves. When a user leaves home for work, the tracking service removes the *user* tag from device adapters on the home PC and adds them to adapters on her smartphone. When she arrives at work, the tracking service removes the user tag from her smartphone and add them to adapters on her work PC. The user tracking service relies on device interactions. When a user interacts with a device, it updates the tags of all sensors on the device to include the user's tag.

Finally, changes in coverage tags (e.g., due to user movements) or device availability (e.g., device disconnections and re-connections) will result in the coordinator reselecting devices for requested inferences and recreating the graph accordingly.

3.2 Inference Partitioning for Efficiency

Beam uses the inference graph for partitioning computation across devices and optimizing for efficiency.

Graph creation and partitioning: The Beam coordinator maintains a set of inference graphs in memory as an *incarnation*. When handling an inference request, the coordinator first incorporates the requested inference graph into the incarnation, re-using already running modules, and merges inference graphs if needed. Once the coordinator finishes resolving all required inputs for each module in the inference graph, it determines where each module should run using the optimization schemes described next. The coordinator then initializes remote channels and partitions the graph into engine-specific subgraphs which are sent to the engines. Whenever the tracking service updates coverage tags, e.g. due to user movements, the coordinator re-computes the inference graphs and sends updated subgraphs to the affected engines. Next, the engines receive their respective subgraphs, compare each subgraph to existing ones, and update them by terminating deleted channels and modules before initializing new ones. Engines ensure that exactly one inference module of each type with a given coverage tag is created.

Optimizing resource usage: In Beam, optimizations are either performed *reactively*, i.e., when an application issues/cancels an inference request, or *proactively* at periodic intervals.

Beam's default reactive optimization determines where each module should run by partitioning the inference graph to minimize the number of remote channels. Let $G(V,E)$ be an inference graph, where V represents the nodes (inference modules), and E represents its adjacency matrix. In E, e_{ij} is the cost of the edge (channel) connecting module i to module j; $e_{ij}=0$ if two modules are not connected directly. Beam's optimizer determines potential partitions of the inference graph and picks the partition with the minimum cost. To determine a partition $P_{|V|\times|D|}$, Beam assigns each module $i \in V$ to run on a device $d \in D$. That is, $p_{id}=1$ if module i runs on device d and $p_{id}=0$ otherwise. We define the cost matrix of a partition P of the inference graph as $C_{|D|\times|D|}=P^T EP$, where $c_{d_1 d_2}$ denotes the sum of the cost of all channels from device d_1 to device d_2. Since the reactive optimizer aims at minimizing the number of remote channels, here $e_{ij}=1$ for all connected modules i and j in the graph. An adapter module runs on a device co-located with the sensor, and an application runs on the device requested by the user. Beam solves the following linear program to find P with the minimum cost:

$$\begin{aligned} \text{Minimize} \quad & \sum_{\forall d_1,d_2\in D, d_1\neq d_2} c_{d_1 d_2} \\ \text{subject to} \quad & \sum_{d\in D} p_{id}=1 \qquad \forall i\in V \\ & p_{id}\in\{0,1\} \qquad \forall i\in V,\ \forall d\in D \end{aligned}$$

Beam's default proactive optimization minimizes the amount of data transferred over remote channels by solving the same linear program but using the data rate profile of each edge as e_{ij}. Engines profile their subgraphs, and report profiling data (e.g., per-channel data rate or estimated per-module CPU utilization) to the coordinator periodically. Other potential optimizations can minimize CPU/memory usage at engines, or IDU delivery latency. Beam allows for modular replacement of optimizers. The coordinator applies optimizations by re-configuring inference graphs and remapping the engine on which each inference module runs.

Scatter node optimization: The coordinator further optimizes the inference graph by finding remote channels which have the same writer module, and whose readers reside on a common engine (R_e). For each such set of edges (E), it adds a single remote channel edge from the writer to a new *scatter* node at R_e. The scatter node is then set as the writer for all edges in E, in effect, replacing multiple remote channels with one and reducing the amount of *wide-area* network transfers by a factor of $|E|$.

3.3 Disconnection Tolerance

Beam's remote channels always go through the coordinator and support reader/writer disconnections by using buffers at the coordinator. Thus, a channel is split into three logical components: writer-side, reader-side, and coordinator-side (present only in remote channels). A channel's writer-side and coordinator-side component buffer IDUs. Channels offer two guarantees: i) readers do

Adapter	Inference Module
PC Event	PC Activity
PC Input	PC Occupancy
Phone GPS	Semantic Location
Accelerometer Fitbit	Fitness Activity
Energy Meter (HomeOS)	Appliance Usage
Camera (HomeOS)	Camera Occupancy
PC Mic/Tablet Mic	Mic Occupancy
PC Mic/Tablet Mic	Social Interaction

Table 1: Sample adapters and inference modules.

not receive duplicate IDUs, and ii) readers receive IDUs in FIFO timestamp order. Beam specifies a default size for remote channel buffers but also allows application developers to customize buffer sizes based on deployment scenarios, e.g., network delays and robustness.

Internally, channels assign sequence numbers to IDUs. They are used for reader-writer flow control, and in remote channels for applying back-pressure on the writer-side component when the coordinator-side buffer is full, e.g., when a reader is disconnected. Currently, the writer-side and coordinator-side buffers use the drop-tail policy to minimize data transfer from writer to coordinator in the event of a disconnected/lazy reader (as opposed to drop head). This design implies that after a long disconnection a reader will first receive old inference values followed by recent ones.

Channels and modules do not persist data. If necessary, applications and modules may use a temporal data store, such as Bolt [37], to make inferences durable.

4 Implementation

Our Beam prototype is implemented in C# as a cross-platform portable service that can be used by .NET v4.5, Windows Store 8.1, and Windows Phone 8.1 applications. The Beam inference library has sample implementations for 8 inference modules and 9 adapters (listed in Table 1). It also includes a HomeOS-adapter that allows Beam to leverage various other device abstractions provided by HomeOS [33], such as the camera and energy meter device drivers used by some of our sample inferences. Each Beam module has a single data event queue and a thread to deliver received IDUs (akin to the actor model [22, 26, 29]). All communication between the coordinator and engine instances uses the SignalR [17] library, and Json.NET [10] is used for data serialization. The engine library, coordinator, sample adapters, and tracking service are implemented in 6614, 952, 1824, and 219 (total=9609) source lines of code respectively.

4.1 Sample Applications

We implement the motivating applications described in Section 2.1 in Beam. Inference graphs of Rules and Quantified Self (QS) are shown in Figure 3 and Figure 2, respectively. Device adapters such as Microphone, Camera, and PC Event adapters are shared by both inference graphs. For common inference modules such as the PC Activity inference, Beam instantiates only one of them across these graphs. Changes in coverage tags and device availability caused by user mobility prompt Beam to re-select appropriate devices for inference graphs. For instance, PC Activity for QS might either be drawn from the home PC or the work PC depending on the user's current location.

4.1.1 Rules Application

The Rules application requires the Appliance Usage and Home Occupancy inferences implemented as follows.

The *Appliance Usage* inference module reads aggregated power consumption of a home from a whole-home power meter, or a utility smart-meter, and disaggregates it to determine the set of appliances that are on at any given instant, using the CO algorithm from [39], configured with 10 commonly owned home appliances [25]. The whole-house power readings are generated using our power-sensor adapter, which interfaces with an Aeon ZWave whole-house meter [1].

The *Mic Occupancy* inference module reads audio samples using the PC Microphone adapter at a sampling rate of 8 kHz (in 4 second frames), and filters out background noise (such as wind, fans, etc.) [38]. If after filtering, the audio sample still indicates sound is present, the inference output is 'occupied'.

The *PC Activity* module infers the current activity a user is performing on a PC (described in Section 4.1.2).

The *Camera Occupancy* module receives streaming video input from an adapter provided by the HomeOS web-cam driver. The input video is of 640×480 resolution and streams at a frame rate of 1 fps. The module compares consecutive frames in the video. If any significant difference indicating possible human movement is detected [28], the inference output is 'occupied'.

The *Home Occupancy* module combines Mic Occupancy, Camera Occupancy, and PC Activity modules, to produce a Home Occupancy inference, outputting 'occupied' if one of the following is true: Mic Occupancy, Camera Occupancy, or PC Activity $\neq$ No activity.

4.1.2 Quantified Self (QS) Application

QS tracks a user's fitness activities, social behaviors, and computing activities on a PC. It is implemented as a Windows Azure web application. Users view plots of their data at leisure on the QS webpage. The inference modules used by this application are described as follows.

The *Social Interaction (Is Alone)* module detects the presence of human voice, outputting 'user not alone' when human voice is present (likely due to conversations with others, though false positives may arise due to TV sounds and background noises). It computes the mel-

```
1 // Inference developers implement module logic
2 public class ModHomeOccupancy:InferenceModuleBase {
3   // Read parameters from the specification XML file
4   public override void Initialize(ModuleSpec spec) {
5     this.paramList = spec.getControlParams();
6     // set state and initialize using parms ...
7   }
8   // Callback to receive IDUs from input channel(s)
9   public override void DataReceived(IChannel channel
        , List<IIDU> inputSignals) {
10    // Compute occupancy based on input
11    HomeOccupancyIDU inferenceResult =
12                computeOccupancy(inputSignals);
13    // Push result IDUs to output channel(s)
14    if (!changedSinceLastPush(inferenceResult))
15      foreach (IChannel ch in outputChannels)
16        ch.Push(inferenceResult);
17  }
18  // ...
19 }
20 // App developer: request inferences from engine
21 public class QSApp : InferenceModuleBase {
22   void startInference() {
23     // Get an instance of the local engine
24     Beam.Engine engine = Beam.Engine.Instance;
25     // Prepare coverage tags
26     List<CoverageTag> tag = new List<CoverageTag>();
27     tag.Add(new PersonCoverageTag("User1"));
28     // Register for inference notifications
29     engine.Request(Beam.Modules.ModHomeOccupancy, tag
          , Mode.FreshPush, this);
30   }
31   // Callback to receive IDUs from input channel(s)
32   public override void DataReceived(IChannel channel
        , List<IIDU> occupancyInferences ) {
33     // Perform actions based on IDUs received ...
34   }
35 }
```

Listing 2: Example usage of the Beam API.

frequency cepstral coefficients (MFCC) [32, 52] over a 200 ms window of the microphone adapter data at 44.1 kHz and uses a decision tree [58] to classify if human voice is present. The module also incorporates movement detection by analyzing video streams from the camera.

The *PC Activity* inference module reads the name of the currently active desktop window from the PC-event adapter using a Win32 system call. It then classifies the name into one of the known PC activity categories (coding, web browsing, social networking, emailing, reading etc.) using a pre-configured mapping. It also infers the psychological state of the user (bored vs. focused) using the features proposed in [51], including window switches, web page switches, time spent browsing Facebook.com, and time spent using e-mail.

The *Fitness Activity* module implements the algorithm from [61] to infer human transportation modes (still, walking, driving) using the phone accelerometer. It also uses the Fitbit [5] API to fetch users' FitBit activity logs, and combines it with accelerometer-based inferences.

4.2 APIs

Listings 1 and 2 show how application and inference developers leverage the Beam APIs using the Home Occupancy inference as an example.

Inference developers provide an XML specification for each inference module (Listing 1) configuring its parameters as well as the input and output channel IDU types. They then implement the module using Beam's APIs (Listing 2, line 1-19) extending the `InferenceModuleBase` helper class. The module is first initialized with control parameters (line 5). It receives inputs in the `DataReceived` callback (line 9), performs the implemented inference logic (line 11), and sends result IDUs to output channels (line 14-16).

Application developers simply request a specific inference module, e.g. Home Occupancy (Listing 2, line 20-35). The application specifies coverage tags (line 26-27), and invokes the local engine's `Request` method (line 29) to register for inference notifications. Beam then instantiates the required inference graph and returns a channel to the application with the requested module as writer. Result IDUs are received by the application via the `DataReceived` callback (line 32).

Application components and their description
Sensor driver: *Handled by M-Hub and Beam* One driver per sensor type.
Inference logic: *Handled by M-Lib and Beam* For each inference an application requires, at least one inference component is needed, e.g., incorporating feature extraction techniques, inference algorithm, learning model, etc.
Parameter tuning: *Simplified by Beam* An application must also incorporate logic to match its inference logic with the underlying sensors (for a range of sensors), e.g. configuring sensor-specific parameters such as sampling rate, frame rate for cameras, sensitivity level for motion sensors, etc.
Cloud service: *Simplified by Beam* Depending on the development approach, an application may require several cloud services, e.g., a storage service for data archival, an execution environment for hosting inference logic, authentication services, etc.
Device disconnection tolerance: *Handled by Beam* Since devices such as smartphones, tablets, may have intermittent connectivity, developers need to appropriately handle disconnections.
User interface (UI): *Simplified by Beam* Typical applications require certain UI components, e.g., to allow configuration of sensors for data collection, or for users to view results.

Table 2: Components of inference-based applications.

5 Evaluation

We evaluate how Beam's inference graph abstraction simplifies application development, benchmark its performance, and evaluate its efficacy in addressing the three key challenges identified in Section 1. Our evaluation uses micro-benchmarks as well as the two motivating applications from Section 4.1.

First, in Section 5.2 we quantify how Beam's abstractions simplify application development and evaluate the overhead of graph creation. Then, in Section 5.3, we evaluate how Beam's device selection in a real-world deployment with user mobility improves inference accuracy. Next, in Section 5.4, we show the impact of Beam's inference graph partitioning to optimize for efficient resource usage. Finally, in Section 5.5 we showcase Beam's ability to handle device disconnections.

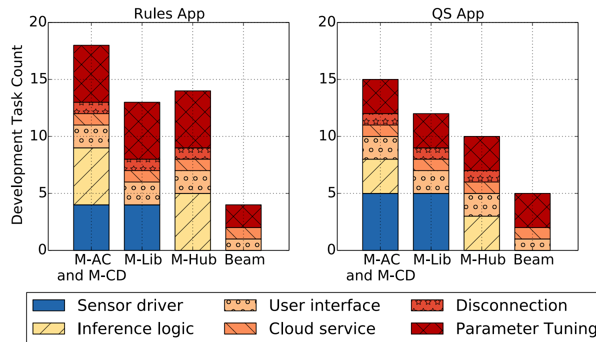

Figure 5: Development tasks using different development approaches in the two applications (Rules, QS)

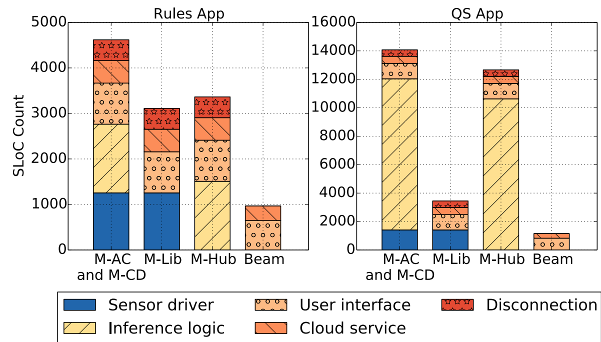

Figure 6: SLoC using different development approaches in the two applications (Rules, QS)

For our experiments, the Beam coordinator runs on a Windows Azure VM (with AMD Opteron Processor, 7 GB RAM, 2 virtual hard disks, running Windows Server 2008 R2); the engines run on desktop machines (with AMD FX-6100 processor, 16 GB RAM, running Windows 8.1) and a Windows Phone (Nokia Lumia). Both sample applications, Rules and Quantified Self (QS), run on the same VM as the coordinator; local engines run in the cloud, a home PC, phone, and a work PC.

5.1 Development Approaches

To quantify the reduction in development effort achieved by Beam, we explore different approaches that a developer may adopt to design such applications.

Monolithic-All Cloud (M-AC). In this approach, the application is developed as a monolithic silo without the use of any framework. All application logic is tightly coupled to the sensing devices, and all collected data is relayed to cloud services, as is the case with Xively [20] and SmartThings [19]. The cloud service runs the application's data processing and inference logic.

Monolithic-Cloud and Device (M-CD). In this approach, an application developer hard-codes the division of inference logic across the cloud VM and end devices [13, 69]. Thus, sensor values are processed to some degree on the end device before being uploaded to the cloud VM which hosts the remainder of the application logic. Depending on the deployment and resource constraints, the developer may need to hand-optimize the resource usage (e.g., CPU, memory, or network usage).

Monolithic-using inference libraries (M-Lib). This approach is similar to the previous one (M-CD), except that application developers may use libraries of inference algorithms tuned by domain experts, thus leading to some reduction in development effort [31, 44, 57].

Monolithic-using sensor hub systems (M-Hub). Platforms such as HomeOS [33], Homeseer [7], and others [15], facilitate the development of applications by providing homogeneous device-based programming abstractions. Typically, these platforms implement sensor drivers and regulate access to different sensors; applications still implement inference logic.

Beam. In this approach, an application on any of the user's devices simply presents its inference requests to the local Beam instance. Using the inference graph abstraction, Beam bears the onus of device selection, optimizing for efficiency, and handling disconnections. Note that using Beam does not preclude the M-Hub approach where all sensing and inference logic run locally on a single hub device (e.g., a home hub). We refer to such scenarios built using Beam's inference abstractions as Beam-Hub, with the engine and coordinator running locally without needing an external network connection.

5.2 Evaluation of Inference Abstraction

In this section we highlight the saving in application development effort using Beam's inference graph abstraction and quantify the overhead of graph creation.

5.2.1 Comparison of Development Effort

We implement our representative applications using the different development approaches described above and present a quantitative comparison of the development effort using two metrics: (i) number of development tasks and (ii) number of source lines of code (SLoC). *Number of development tasks* is defined as the number of architectural components that need to be designed, implemented, and maintained for a complete functioning application. To analyze development effort in greater depth, these components can further be categorized based on the function they perform (Table 2). This metric captures the diverse range of tasks developers of applications for connected devices are required to handle. Although comparing the number of tasks provides insight into the development effort required for each approach, different components often require varying levels of implementation efforts. Thus, to distinguish individual components, we also measure the *number of source lines of code (SLoC)* required for the components in each approach.

Figures 5 and 6 show the number of development tasks and number of SLoC, respectively, for the Rules and QS

	Sample scenario 1 (local inference)		Sample scenario 2 (remote inference)		
	App1's request	App2's request	App1's request	App2's request	Reevaluation
Total	232.54 $\pm$ 1.63	246.71 $\pm$ 16.62	237.43 $\pm$ 12.76	230.24 $\pm$ 3.53	-
Request and subgraph transfer	230.35 $\pm$ 1.68	246.24 $\pm$ 16.62	236.13 $\pm$ 12.75	229.73 $\pm$ 3.47	-
Coordinator (inference graph creation)	1.05 $\pm$ 0.14	0.16 $\pm$ 0.01	0.90 $\pm$ 0.04	0.20 $\pm$ 0.07	0.12 $\pm$ 0.01
Coordinator (split inference graphs)	0.06 $\pm$ 0.01	0.12 $\pm$ 0.01	0.06 $\pm$ 0.01	0.15 $\pm$ 0.07	0.11 $\pm$ 0.01
Engine (instantiate subgraphs)	1.05 $\pm$ 0.13	0.16 $\pm$ 0.03	0.30 $\pm$ 0.08	0.12 $\pm$ 0.04	0.40 $\pm$ 0.10

Table 3: Inference graph setup times (in ms) in two sample scenarios, with one standard deviation.

applications using the different development approaches. We observe that for the Rules application, Beam reduces the number of development tasks by 4.5×, and the number of SLoC by 4.8×, compared with M-AC and M-CD. Similarly, for the QS application, Beam reduces the number of development tasks by 3×, and the number of SLoC by 12×, compared with M-AC and M-CD.

Number of development tasks: As shown in Figure 5, the approaches of Monolithic-All Cloud (M-AC) and Monolithic-Cloud and Device (M-CD) have similar number of development tasks for both the Rules (on left) and the QS application (on right). M-CD requires developers to hard-code the division of tasks between endpoint devices and cloud servers, thus statically optimizing for better resource usage than M-AC (Section 5.4).

Compared with M-AC and M-CD, the M-Lib approach reduces developer effort. It leverages existing libraries which provide implementations of inference algorithms and also handle their training and tuning. Similarly, in the M-Hub approach, developer effort is reduced due to existing sensor driver implementations provided by the platform. Finally, when using Beam, application developers do not need to design or implement sensor drivers, inference logic, tuning timing parameters, or handling disconnections. Application developers only need to decide their required inferences, and develop application-specific components, e.g., user interface, third-party authentication, etc.

Number of SLoC: As shown in Figure 6, we observe that for all approaches, the SLoC count is generally proportional to the development task count. For most approaches SLoC is dominated by tasks of developing sensor drivers and inference logic. For instance, the Social Interaction inference in QS contributes more than 9796 SLoC. Both Beam and M-Lib help alleviate this complexity. Beam improves upon M-Lib by handling the complexity of implementing sensor drivers, disconnection tolerance, and optimizing resource usage, etc.

5.2.2 Overhead of Inference Graph Creation

We study the time taken by Beam to satisfy requests for a single Mic Occupancy inference, which in turn uses the PC Mic adapter. We consider two sample scenarios, 1) applications request for a local inference, and 2) applications request for a remote inference. In both cases, application 1 initiates a request first, followed by application 2, with the same coverage tag.

In both scenarios, the overhead of instantiating and maintaining the inference graph at end-points is minimal and dwarfed by the latency of transferring the request to the coordinator and receiving back the subgraphs.

Table 3 shows the overhead of graph creation for each of the scenarios. In both cases, the second request uses less time for graph creation at the coordinator, since much of the graph already exists when the second request arrives (e.g., module specifications are not re-read). Likewise, in both scenarios, time spent at the engine(s) in applying the subgraph is lower for the second request as compared to the first request. Further, it is lower in scenario 2 because the inference graph is split across two engines. Lastly, the coordinator performs a periodic re-evaluation based on the channel data rates and applies the proactive optimization discussed in Section 3.2. The time taken to perform the re-evaluation is minimal.

5.3 Device Selection

Unlike other approaches described in Section 5.1, the inference graph in Beam can select devices for applications even in heterogeneous environments with user mobility, resulting in increased inference accuracy. We demonstrate this using the PC Activity inference in the context of the QS application (inference graph in Figure 2).

We perform an experimental lab deployment with two locations - a lab which acts as 'home' and an office. Movements from home to office are used to simulate user commuting. We compute Beam's inference accuracy against manually-collected ground truth data from the deployment, and compare it to three other development approaches that may be used in the absence of a Beam-like tracking service. The first approach performs the PC Activity inference using only inputs from the home PC, while the second approach uses only inputs from the work PC. We assume that the home PC goes into sleep after a certain period of user inactivity, while the work PC remains on even after the user leaves. In the third approach, the inference is drawn using simultaneous inputs from both the home and work PCs. However, when the two inputs conflict, the output is set to 'Other'.

Figure 7 shows a comparison of inference accuracy for these different schemes over a ten minute interval of using the QS application. Inferring PC-based activities using only the home PC works accurately until the user

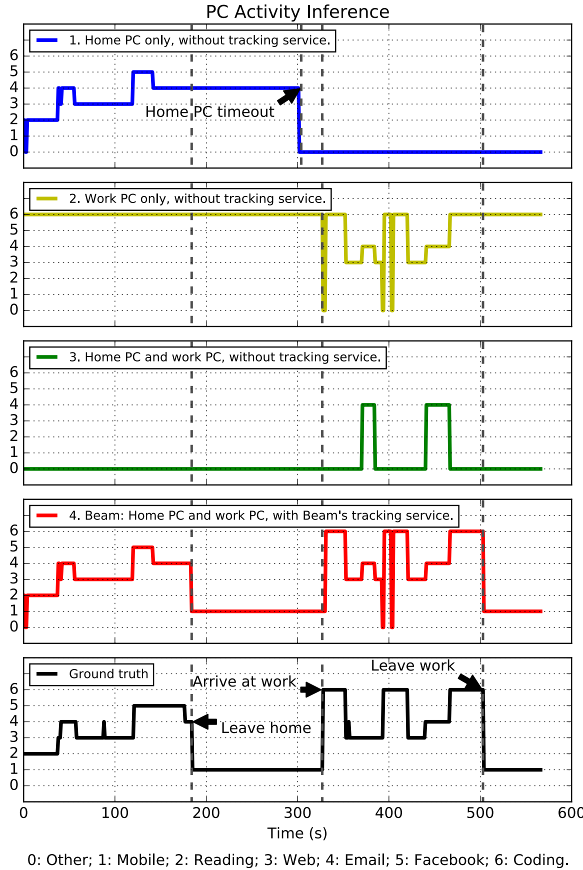

Figure 7: Beam's tracking service improves inference accuracy (measured against ground truth) significantly over other approaches all of which fail to select devices in the presence of user mobility.

leaves home, but deviates significantly from ground truth once the user has left. Similarly, using only the work PC can only accurately compute the PC-based activities of the user after the user arrives at work. On the other hand, using both the work and home PC without a tracking service often produces conflicting results, for instance, when home PC and work PC both generate PC Activity inferences during user commuting. Beam's tracking service correctly identifies the location of the user and triggers the inference graph to re-select appropriate devices, achieving inference accuracy 3× higher than the best performing scheme above. Using the tracking service, Beam's smartphone engine can also correctly indicate that the user is 'Mobile' while commuting. Table 4 summaries these accuracy improvements.

Although the above experiments are performed in a lab setting with a simulated commuting scenario, having a longer commuting time will only reduce the accuracy of non-Beam approaches, since only Beam with the tracking service can infer user commuting and all other approaches will yield incorrect results. Finally, we expect to observe a similar accuracy improvement for other

Setup	Accuracy
Home PC only, without tracking service	29.68%
Work PC only, without tracking service	26.94%
Home PC and work PC, without tracking service	4.59%
Home PC and work PC, with Beam's tracking service	88.16%

Table 4: Accuracy of PC Activity Inference compared to ground truth (a summary of Figure 7).

inferences that require handling of sensor coverage, e.g. the Social Interaction inference in the QS application.

5.4 Efficient Resource Usage

Next, we illustrate that Beam can match the resource usage of hand-optimized applications by partitioning the inference graph across devices. We also evaluate different optimization schemes used in Beam. Although we consider network usage to benchmark Beam in this paper, we expect similar optimizations can be performed on other resources such as CPU usage, latency, and energy.

Graph partitioning: For the Rules and QS applications, we compare Beam's data transfer overhead (i.e., number of bytes transferred over the wide area) with that of different approaches (M-AC, M-CD, M-Lib, M-Hub). Figure 8 shows the total number of bytes transferred over the wide area in one hour, for the sample applications using different approaches. Medians and standard deviations across three runs are reported. M-AC incurs the largest overhead, because it transfers all sensor data from the device to a cloud VM for processing. On the other hand, the M-CD, M-Lib, and M-Hub approaches are optimized to perform most of their processing at the edges before transferring data to the cloud VM. Beam automatically partitions the inference graph using both reactive and proactive optimizations and comes close to matching the network transfer overhead incurred by M-CD; it incurs a slightly higher overhead for transferring *control messages* such as forwarding the application's request to the coordinator, receiving the part of the inference graph to instantiate, sending channel data rates to coordinator (for proactive optimization), acknowledgments, etc. Note that, when the M-Hub approach is used for the Rules application, there is no wide area IDU transfers because all

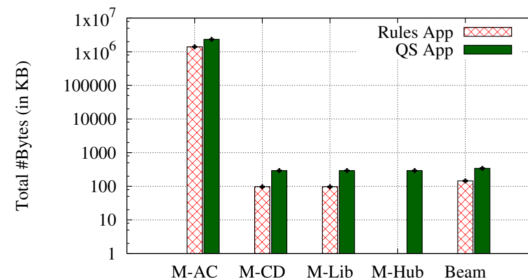

Figure 8: Total bytes transferred over the wide area for a 60 minute run of the Rules and QS apps using different approaches. Y-axis is in log scale.

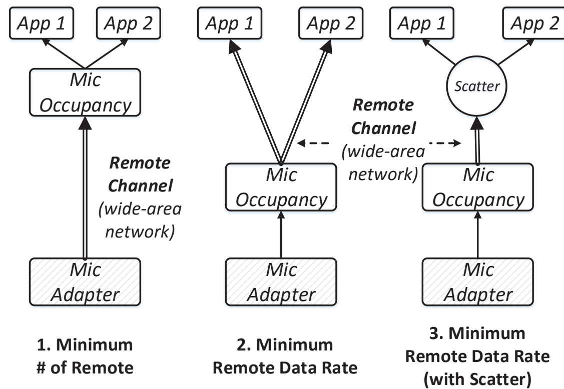

Figure 9: Sample configurations of the Mic Occupancy inference, with different optimization goals.

required sensors are present locally at home.

Optimization schemes: Next, we evaluate the effect of different optimization schemes in Beam. We focus on a simple inference graph, where two applications running on cloud servers subscribe to the Mic Occupancy inference. Figure 9 shows the three configurations that result from Beam optimizations in isolation and Figure 10 shows their network resource consumption over a 100-second interval. Beam's default reactive optimization (Figure 9 #1) minimizes the number of remote channels resulting in a large amount of microphone data being transmitted over the wide area. Beam's proactive optimization notices these large uploads and uses channel data rates to re-evaluate and re-partition the inference graph (Figure 10 at 20 s), thus moving the Mic Adapter closer to the edge (Figure 9 #2), and reducing wide area transfers significantly. Finally, enabling Beam's scatter node optimization (Figure 9 #3) halves the network overhead, for two consumer applications, compared with the proactive optimization without the scatter node.

5.5 Handling Disconnections

In this section we quantify the ability of the Beam inference graph to handle device disconnections. Remote

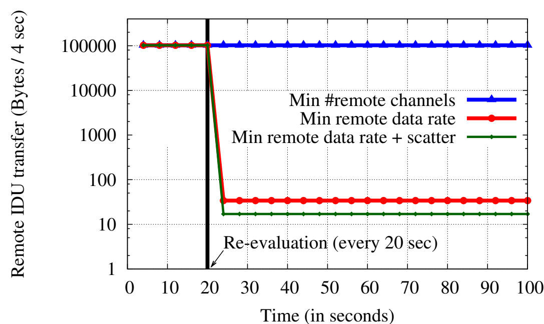

Figure 10: Network resource consumption over a 100 seconds interval for configurations in Figure 9. Y-axis is in log scale. IDUs are generated every 4 seconds.

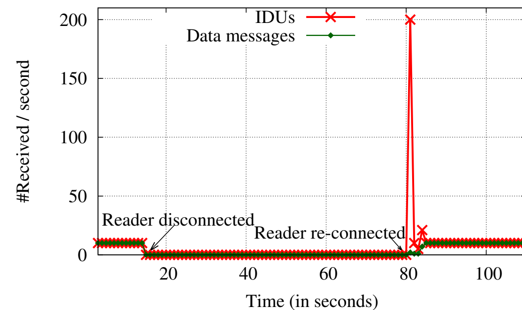

Figure 11: Remote channel time trace. Write rate is 10 values per second, and writer buffer and coordinator buffer are sized at 100 values each.

channels in Beam buffer data at both the coordinator and at the writer endpoints to tolerate reader and writer disconnections. The size of these buffers and the writer's sending rate determine the time window for which disconnections are lossless, and can be sized as per the deployment scenario. Figure 11 shows a time series plot of the number of IDUs and data messages received by a reader over a 100 seconds interval. The writer produces ten IDUs every second. Each IDU produced is pushed out in a separate data message until a reader disconnection at t=15s results in data buffering, first at the coordinator, and then at the writer. We constrain the channel buffers at the writer and coordinator ends to 100 IDUs each, thus supporting buffering of only 20 seconds worth of IDUs in this configuration, forcing the remaining IDUs to be dropped. When the reader reconnects at t=80 s, the 200 buffered IDUs are batched in a small number of data messages and delivered to the reader, showing Beam's support for tolerating device disconnections.

6 Related Work

Beam's inference graph draws inspiration from dataflow graphs used in a wide range of scenarios such as routers [45], operating systems [54, 62], data-parallel computation frameworks [6, 40], and Internet services [70]. Beam is the first framework that provides inference abstractions to decouple applications, inference algorithms, and devices, using the inference graph for device selection, efficiency, and disconnection tolerance. We classify prior work into four categories.

Device abstraction frameworks: HomeOS [33] and other platforms [7, 15, 21, 68] provide homogeneous programming abstractions to communicate with devices. For instance, HomeOS applications can use a generic motion sensor role, regardless of the sensor's vendor and protocol. These approaches only decouple device-specific logic from applications, but are unable to decouple inference algorithms from applications. Moreover, they cannot provide device selection or inference partitioning capabilities.

Cross-device frameworks: Rover [41], an early distributed object programming framework for mobile applications, allows programmers to partition client-server applications; it provides abstractions such as relocatable objects and queued remote procedure calls to ease application development. Sapphire [73], a more recent framework, requires programmers to specify per-object deployment managers which aid in runtime object placement decisions, while abstracting away complexities of inter-object communication. MagnetOS [48] dynamically partitions a set of communicating Java objects in a sensor network with a focus on energy efficiency. Like these frameworks, channels in Beam abstract away local and remote inter-module communication. Beam fundamentally differs from them by using the inference graph to decouple applications from sensing and inferences, aid in device selection to operate in heterogeneous environments, and support global resource optimizations.

Macro-programming frameworks [27, 36, 49] provide abstractions to allow applications to dynamically compose dataflows [50, 56]. Semantic Streams [71] and Task Cruncher [66] address sharing sensor data and processing across devices. However these approaches focus on data streaming and simple processing methods, e.g., aggregations, rather than generic inferences, and do not target general purpose devices e.g., phones, PCs. In addition, they do not address device selection or inference partitioning at runtime.

Mobile sensing frameworks: Existing work has focused only on applications requiring continuous sensing on a *single* mobile device. Kobe [31], Auditeur [57], and Senergy [44] propose libraries of inference algorithms to promote code re-use and explore single device energy-latency-accuracy trade-offs. Other work [42, 43, 44, 47] has focused on improving resource utilization by sharing sensing and processing across multiple applications on a mobile device. None of these approaches address problems such as modular inference composition, device selection with user mobility, inference partitioning across multiple devices, or handling disconnections.

An early version of our work appears in a workshop paper that outlined the problem and presented a basic design [65]. The current paper extends the design, implements real applications, and evaluates performance.

7 Discussion

We discuss potential improvements to Beam.

Error and error propagation: Beam currently supports typed errors such as probability distributions (e.g. mean and standard deviation), and error margin (e.g. center and radius). Although error propagation has been studied in the field of artificial intelligence (e.g. neural network [63]), there is no prior work on error propagation in mobile context sensing. We are investigating techniques to enable inference module developers to implement customized error propagation functions for specific inferences, so that Beam can propagate the error from a module's inputs to its output.

Actuation and timeliness: Many in-home devices possess actuation capabilities, such as locks, switches, cameras, and thermostats. Applications and inference modules in Beam may want to use such devices. If the inference graph for these applications is geo-distributed, timely propagation and delivery of such actuation commands to the devices becomes important and raises interesting questions of what is the *safe* thing to do if an actuation arrives "late".

Data archival and correlation mining: Prior work has shown that exploiting the correlation among inferences can effectively reduce sensing cost [55]. While Beam modules do not currently store data either at the engines or the coordinator, applications and modules may use a temporal datastore, such as Bolt [37], to make inferences durable. Storing and querying archived inference data will allow inference developers to perform correlation mining to improve inferences.

Data privacy: While we do not address privacy concerns in our work, we believe the use of inferences can enable better data privacy controls [30]. For example, users may allow an application to access the occupancy inference (using a camera) instead of the raw image data used for drawing the inference. This prevents the leakage of private information by preventing other inferences that can be drawn using the raw data. Moreover, Beam's coverage tags allow the user to define fine-grained controls, for instance, allowing an application to access activity inference only for a certain user tag.

8 Conclusion

Applications using connected sensing devices are difficult to develop today because they must incorporate all the data sensing and inference logic, even as devices move or are temporarily disconnected. We design and implement Beam, a framework and runtime for distributed applications using connected devices. Beam introduces the inference graph abstraction which is central to decoupling applications, inference algorithms, and devices. Beam uses the inference graph to address the challenges of device selection, efficient resource usage, and device disconnections. Using Beam, we develop two representative applications (Rules and QS), where we show up to 4.5× lower number of tasks and 12× lower source line of code in application development effort, with negligible runtime overhead. Moreover, Beam results in up to 3× higher inference accuracy due to its ability to select devices in heterogeneous environments, and Beam's dynamic optimizations match hand-optimized applications for network resource usage.

References

[1] Aeon Labs Z-Wave Smart Energy Switch. http://aeotec.com/.

[2] Amazon Home Automation Store. http://www.amazon.com/home-automation-smarthome/b?ie=UTF8&node=6563140011.

[3] The US Market for Home Automation and Security Technologies. Technical report, BCC Research, IAS031B, 2011.

[4] Dropcam - Super Simple Video Monitoring and Security. https://www.dropcam.com/.

[5] Fitbit. https://www.fitbit.com/.

[6] Google Cloud Dataflow. https://cloud.google.com/dataflow/.

[7] HomeSeer. http://homeseer.com/.

[8] iOS: Understanding iBeacon. http://support.apple.com/kb/HT6048.

[9] IFTTT: Put the internet to work for you. https://ifttt.com/.

[10] Json.NET. http://www.newtonsoft.com/json.

[11] Map my fitness. http://www.mapmyfitness.com/.

[12] Nest. http://www.nest.com/.

[13] Optimized App. http://optimized-app.com/.

[14] Quantified self. http://quantifiedself.com/.

[15] Revolv. http://revolv.com/.

[16] ShopKick shopping app. http://shopkick.com/.

[17] ASP.NET SignalR. http://signalr.net/.

[18] Simplicam Home Surveillance Camera. https://www.simplicam.com/.

[19] SmartThings. http://www.smartthings.com/.

[20] xively by LogMein. https://xively.com/.

[21] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong. Rio: A system solution for sharing I/O between mobile systems. In *Proc. ACM MobiSys*, 2014.

[22] J. Armstrong. Erlang. *CACM*, 53:68–75, Sept 2010.

[23] E. Arroyo, L. Bonanni, and T. Selker. Waterbot: Exploring feedback and persuasive techniques at the sink. In *Proc. ACM CHI*, 2005.

[24] R. K. Balan, K. X. Nguyen, and L. Jiang. Real-time trip information service for a large taxi fleet. In *Proc. 9th ACM MobiSys*, June 2011.

[25] N. Batra, J. Kelly, O. Parson, H. Dutta, W. J. Knottenbelt, A. Rogers, A. Singh, and M. Srivastava. NILMTK: An open source toolkit for non-intrusive load monitoring. In *Proc. ACM e-Energy*, 2014.

[26] P. A. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, March 2014.

[27] A. Boulis, C.-C. Han, R. Shea, and M. B. Srivastava. Sensorware: Programming sensor networks beyond code update and querying. *Pervasive Mob. Comput.*, 3(4):386–412, Aug. 2007.

[28] A. B. Brush, J. Jung, R. Mahajan, and F. Martinez. Digital neighborhood watch: Investigating the sharing of camera data amongst neighbors. In *Proc. ACM CSCW*, 2013.

[29] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *Proc. ACM SOCC*, 2011.

[30] S. Chakraborty, C. Shen, K. R. Raghavan, Y. Shoukry, M. Millar, and M. Srivastava. ipShield: A framework for enforcing context-aware privacy. In *Proc. 11th USENIX NSDI*, April 2014.

[31] D. Chu, N. D. Lane, T. T.-T. Lai, C. Pang, X. Meng, Q. Guo, F. Li, and F. Zhao. Balancing energy, latency and accuracy for mobile sensor data classification. In *Proc. 9th ACM SenSys*, Nov. 2011.

[32] S. B. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 28(4):357–366, 1980.

[33] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *Proc. 9th USENIX NSDI*, Apr. 2012.

[34] D. Evans. The Internet of Things: How the next evolution of the Internet is changing everything. *CISCO white paper*, 2011.

[35] J. Froehlich, L. Findlater, M. Ostergren, S. Ramanathan, J. Peterson, I. Wragg, E. Larson, F. Fu, M. Bai, S. Patel, and J. A. Landay. The design and evaluation of prototype eco-feedback displays for fixture-level water usage data. In *Proc. ACM CHI*, 2012.

[36] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using Kairos. In *Distributed Computing in Sensor Systems*, pages 126–140. Springer, 2005.

[37] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan. Bolt: Data management for connected homes. In *Proc. 11th USENIX NSDI*, Apr. 2014.

[38] T. Hao, G. Xing, and G. Zhou. isleep: Unobtrusive sleep quality monitoring using smartphones. In

Proc 11th ACM SenSys, 2013.

[39] G. Hart. Nonintrusive appliance load monitoring. *Proceedings of the IEEE*, 1992.

[40] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. EuroSys*, Mar. 2007.

[41] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *Proc. 15th ACM SOSP*, Dec. 1995.

[42] Y. Ju, Y. Lee, J. Yu, C. Min, I. Shin, and J. Song. Symphoney: A coordinated sensing flow execution engine for concurrent mobile sensing applications. In *Proc. 10th ACM SenSys*, Nov. 2012.

[43] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song. Seemon: Scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *Proc. ACM MobiSys*, 2008.

[44] A. Kansal, S. Saponas, A. B. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola. The Latency, Accuracy, and Battery (LAB) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing. In *Proc. ACM OOPSLA*, Nov. 2013.

[45] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[46] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis. Design and deployment of industrial sensor networks: Experiences from a semiconductor plant and the North Sea. In *Proc. 3rd ACM SenSys*, Nov. 2005.

[47] F. Lai, S. S. Hasan, A. Laugesen, and O. Chipara. Csense: A stream-processing toolkit for robust and high-rate mobile sensing applications. In *Proc. 13th ACM/IEEE IPSN*, 2014.

[48] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. G. Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *Proc. ACM MobiSys*, 2005.

[49] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(3):543–576, 2006.

[50] G. Mainland, M. Welsh, and G. Morrisett. Flask: A language for data-driven sensor network programs. *Harvard Univ., Tech. Rep. TR-13-06*, 2006.

[51] G. Mark, S. T. Iqbal, M. Czerwinski, and P. Johns. Bored Mondays and focused afternoons: The rhythm of attention and online activity in the workplace. In *Proc. 32nd ACM CHI*, April 2014.

[52] P. Mermelstein. Distance measures for speech recognition, psychological and instrumental. *Pattern recognition and artificial intelligence*, 116: 374–388, 1976.

[53] D. Morris, A. B. Brush, and B. R. Meyers. Superbreak: Using interactivity to enhance ergonomic typing breaks. In *Proc. ACM CHI*, 2008.

[54] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proc. 2nd USENIX OSDI*, Oct. 1996.

[55] S. Nath. Ace: Exploiting correlation for energy-efficient and continuous context sensing. In *Proc. ACM MobiSys*, 2012.

[56] R. Newton, G. Morrisett, and M. Welsh. The Regiment macroprogramming system. In *Proc. 6th ACM/IEEE IPSN*, 2007.

[57] S. Nirjon, R. F. Dickerson, P. Asare, Q. Li, D. Hong, J. A. Stankovic, P. Hu, G. Shen, and X. Jiang. Auditeur: A mobile-cloud service platform for acoustic event detection on smartphones. In *Proc. 11th ACM MobiSys*, June 2013.

[58] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[59] M. M. Rahman, A. A. Ali, K. Plarre, M. al'Absi, E. Ertin, and S. Kumar. mConverse: Inferring conversation episodes from respiratory measurements collected in the field. In *Proc. 2nd Wireless Health*. ACM, 2011.

[60] H. Ramamurthy, B. S. Prabhu, R. Gadh, and A. Madni. Wireless industrial monitoring and control using a smart sensor platform. *Sensors Journal, IEEE*, 7(5):611–618, May 2007.

[61] S. Reddy, M. Mun, J. Burke, D. Estrin, M. Hansen, and M. Srivastava. Using mobile phones to determine transportation modes. *ACM Transactions on Sensor Networks (TOSN)*, 6(2):13, 2010.

[62] D. M. Ritchie. A stream input-output system. In *AT&T Bell Laboratories Technical Journal*, 1984.

[63] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5, 1988.

[64] R. P. Singh, S. Keshav, and T. Brecht. A cloud-based consumer-centric architecture for energy data analytics. In *Proc. ACM e-Energy*, 2013.

[65] R. P. Singh, C. Shen, A. Phanishayee, A. Kansal, and R. Mahajan. A case for ending monolithic apps for connected devices. In *Proc. HotOS XV*, May 2015.

[66] A. Tavakoli, A. Kansal, and S. Nath. On-line sensing task optimization for shared sensors. In *Proc. 9th ACM/IEEE IPSN*, 2010.

[67] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman. Practical trigger-action programming in the smart home. In *Proc. ACM CHI*, 2014.

[68] M. Van Kleek, K. Kunze, K. Partridge, et al. Opf: a distributed context-sensing framework for ubiquitous computing environments. In *Ubiquitous Computing Systems*, pages 82–97. Springer, 2006.

[69] R. Wang, F. Chen, Z. Chen, T. Li, G. Harari, S. Tignor, X. Zhou, D. Ben-Zeev, and A. T. Campbell. Studentlife: Assessing behavioral trends, mental well-being and academic performance of college students using smartphones. In *Proc. ACM Ubicomp*, 2014.

[70] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th ACM SOSP*, Oct. 2001.

[71] K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In *Proc. EWSN*, 2006.

[72] D. Wyatt, T. Choudhury, J. Bilmes, and J. A. Kitts. Inferring colocation and conversation networks from privacy-sensitive audio with implications for computational social science. *ACM Trans. Intell. Syst. Technol.*, 2(1), Jan. 2011.

[73] I. Zhang, A. Szekeres, D. Van Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy. Customizable and extensible deployment for mobile/cloud applications. In *Proc. USENIX OSDI*, 2014.