Rivulet: A Fault-Tolerant Platform for Smart-Home Applications

Masoud Saeida Ardekani
Samsung Research America

Rayman Preet Singh*
Amazon

Nitin Agrawal
Samsung Research America

Douglas B. Terry*
Amazon

Riza O. Suminto*
University of Chicago

Abstract

Rivulet is a fault-tolerant distributed platform for running smart-home applications; it can tolerate failures typical for a home environment (*e.g.*, link losses, network partitions, sensor failures, and device crashes). In contrast to existing cloud-centric solutions, which rely exclusively on a home gateway device, Rivulet leverages redundant smart consumer appliances (e.g., TVs, Refrigerators) to spread sensing and actuation across devices local to the home, and avoids making the Smart-Home Hub a single point of failure. Rivulet ensures event delivery in the presence of link loss, network partitions and other failures in the home, to enable applications with reliable sensing in the case of sensor failures, and event processing in the presence of device crashes. In this paper, we present the design and implementation of Rivulet, and evaluate its effective handling of failures in a smart home.

CCS Concepts • **Computer systems organization** → **Distributed architectures**; **Sensors and actuators**; **Reliability**;

Keywords Distributed platform, Fault tolerance, Internet-of-things, Smart homes

ACM Reference Format:
Masoud Saeida Ardekani, Rayman Preet Singh, Nitin Agrawal, Douglas B. Terry, and Riza O. Suminto. 2017. Rivulet: A Fault-Tolerant Platform for Smart-Home Applications. In *Proceedings of Middleware '17*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3135974.3135988

*Work done at Samsung Research America.

Middleware '17, December 11–15, 2017, Las Vegas, NV, USA

ACM ISBN 978-1-4503-4720-4/17/12…$15.00
https://doi.org/10.1145/3135974.3135988

1 Introduction

We are living in an era of "smart" homes. A multitude of solutions [11, 15, 17, 18, 30] are becoming available to transform *legacy* homes into *intelligent* ones, capable of running a wide range of applications ranging from automated thermostat control [36, 58] to identifying a physical intrusion in the home [24]; key to the realization of such solutions is the availability of a diverse range of sensors, actuators, and network-enabled consumer appliances. The smart-home applications themselves operate on *streams* of events generated by the sensors such as a temperature or motion sensor, *actuate*, or control, physical entities such as light bulbs and door locks, and create workflows to automate everyday tasks such as ordering groceries.

The vast majority of existing smart–home solutions are *cloud centric* wherein applications run in the cloud, and need to frequently communicate with in-home devices. Therefore, data from sensors and appliances needs to be sent to the cloud, and the results of computations, including actuation commands, are to be sent back to the home. To facilitate this interaction more efficiently, the solutions require homes to install low-cost gateways (called Smart-Home Hubs or simply *hubs*) which are responsible for communicating with sensor devices using different wireless protocols (e.g., Zigbee [20], ZWave [19], Bluetooth Low Energy [7]).

A cloud-centric solution has several limitations. First, and foremost, all applications are subject to the vagaries of the home Internet connection, particularly the round-trip latency to the cloud and network disruptions. While the cloud-resident application itself may be highly available, delays, faults, and congestion at the ISP directly affect the home applications. Second, privacy is a significant concern for smart-home residents [34], particularly given the sensitive nature of the data (*e.g.*, occupancy). Third, application developers must handle the relatively large volumes of sensor data being transmitted to, stored in, and processed in the cloud, which can be expensive [57].

We advocate a *home-centric* model in which applications rely more heavily on local execution *within* the home; this places greater computational responsibility on the hub and smart consumer appliances. While the home hub alone has limited compute power, and remains the single point of failure, smart consumer appliances are becoming increasingly

capable; TVs [14], refrigerators [9], ovens and washing machines are going to have sufficient computational power to run smart-home applications. Existing solutions fail to leverage this distributed set of computational resources.

To realize a home-centric model, we have built Rivulet, a distributed platform for running smart-home applications on heterogeneous consumer appliances. Rivulet ensures reliable delivery of sensor data and actuation commands to compute devices through a novel *delivery service* and handles different failures for fault-tolerant app execution through an *execution service*.

Smart-home apps process data similar to conventional stream processing. Data (from sensors and appliances) is continuously fed as input to an app for processing; the result of the computation can actuate another in-home device or result in a notification being sent to a user. For example, an app controlling the thermostat receives temperature readings from a sensor and uses them to decide when and how to actuate the HVAC. Rivulet applications can appropriately choose between two delivery guarantees – *Gap*, a best-effort guarantee for delivery of sensor events to applications, and *Gapless*, a stronger delivery guarantee that ensures event delivery despite failures at the cost of bandwidth and battery life. These delivery guarantees, defined post-ingest by Rivulet, are conceptually similar to recovery guarantees introduced by Hwang et al. [40]. An application, e.g., thermostat control, that can afford to miss a few temperature values can choose Gap delivery, while another one that cannot, *e.g.*, intrusion detection, can choose Gapless. This paper makes the following key contributions:

- We study the different types of failures that occur in home environments, the fault-tolerance requirements of smart home applications, and the opportunities in leveraging the natural redundancy of appliances to serve these requirements. Through a survey and a home deployment, we establish that applications differ significantly in their requirements for delivery guarantees for different input sensors.
- We formulate two configurable delivery guarantees for smart home applications: *Gap*, and *Gapless*, and introduce low-overhead protocols to provide those guarantees.
- We extend the traditional stream-processing programming model to provide developers with a declarative approach to express the desired levels of fault tolerance.

Rivulet has been implemented on hardware for smart-home hubs, and Android phones; we performed a feasibility study using a sample home deployment and an array of real sensors and actuators. As part of Rivulet's implementation, we developed Z-Wave and Zigbee adapters in order to communicate with a realistic set of devices popular in smart homes. Our testbed evaluation showed that Rivulet's Gapless and Gap protocols provide fault tolerance at a low overhead, and that applications continued to run in the presence of failures.

2 Challenges & Motivation

In this section, we first outline some of the challenges that exist in home environments, and then present the findings of our study of smart-home applications to motivate the need for the different delivery guarantees.

2.1 Challenges in a Home Environment

A home is not a data center. Smart homes, unlike data centers, are rarely managed by a professional system administrator, and are especially susceptible to errors in management, configuration and device failures. Furthermore, failures are also harder to recover from due to lack of redundant infrastructure and expertise. Conventional distributed system techniques cannot always be applied in a home since many underlying assumptions (e.g., majority of replicas are not faulty) cannot always be guaranteed.

In a study, Hnat et al. [39] deployed over 350 sensors across 20 homes for over 8 months and observed a variety of failures from process failures (due to plug disconnections, hardware failures, and driver crashes), link loss between sensors and hubs (due to concrete slab flooring, copper siding, and radio interference), and sensor failures (due to battery drain and plug disconnections). Moreover, a process downtime of up to 14% was observed, whereas sensor-process link loss and sensor failures occurred for 1-2% of the time. As we discuss in Section 2.2, for certain apps, even short periods of unavailability are undesirable, and therefore a smart-home platform should be able to handle hub failures, link losses, and sensor failures.

Diverse wireless networks. Variety of low-power wireless networks such as ZigBee [20], Z-Wave [19], or BLE [7] exist for sensing devices. Such technologies are prone to message loss and differ significantly in their communication properties, e.g., communication range and multicasting, compared to a typical home WiFi network. The specific choice of network is based on a combination of the desired energy profile and communication range along with the physical form factor; not all hubs can communicate with all kinds of sensors that may be present in a home, leading to cliques of interconnected sensors and hubs. Due to the limited range of aforementioned networks (*e.g.*, 10-20 meters for Zigbee, 40 meters for Z-Wave, and 100 meters for BLE), the physical placement of devices and sensors poses an additional challenge; events from a given sensor may only be reachable to a subset of the processes. The network communication is also subject to radio interference from other home appliances [56] (*e.g.*, microwave ovens, cordless phones) as well as signal degradation from walls and other obstructions [43, 46].

To better understand event loss and duplication in a typical home, we conducted a preliminary study. We deployed six off-the-shelf sensors (four motion and two door sensors) in a home for a period of 15 days. We used Z-Wave sensors configured to multicast events to three Z-Wave processes.

Application	Primary Function	Sensor Type	Type	Delivery Type
Occupancy-based HVAC	Set the thermostat set-point based on the occupancy [58]	Occupancy	Efficiency	Gap
User-based HVAC	Set the thermostat set-point based on the user's clothing level [32]	Camera	Efficiency	Gap
Automated lighting	Turn on lights if user is present, e.g., SmartLights [1]	Occupancy, camera, microphone	Convenience	Gap
Appliance alert	Alert user if appliance is left on while home is unoccupied [60]	Appliance, whole-house energy	Efficiency	Gap
Activity tracking	Periodically infer physical activity using microphone frames [42]	Microphone	Convenience	Gap
Fall alert	Issue alert on a fall-detected event [27, 51, 62]	Wearables [27]	Elder care	Gapless
Inactive alert	Issue alert if motion/activity not detected [1]	Motion, door-open [15]	Elder care	Gapless
Flood/fire alert	Issue alert on a water(or fire) detected event [2]	Water, smoke [4, 12]	Safety	Gapless
Intrusion-detection	Record image/issue alert on a door/window-open event	Door-window [4]	Safety	Gapless
Energy billing*	Update energy cost on a power-consumption event [61]	Energy [4]	Billing	Gapless
Temperature-based HVAC	Actuate heating/cooling if temperature crosses a threshold [36]	Temperature	Efficiency	Gapless
Air (or light) monitoring	Issue alert if CO2/CO level surpasses a threshold [1, 66]	CO, CO2	Safety	Gapless
Surveillance	Record image if it has an unknown object [24]	Camera	Safety	Gapless

Table 1. Desired delivery types for selected example applications.

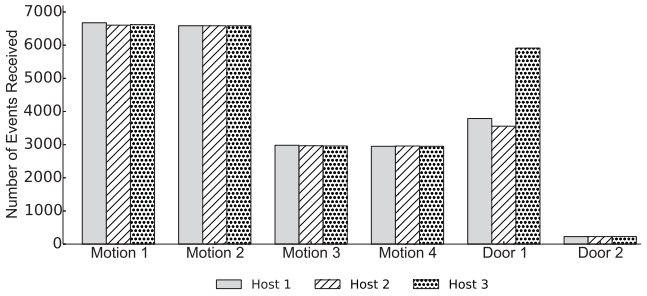


Figure 1. Number of events received at different processes from different sensors in a sample home deployment.

Figure 1 shows the number of events received at each process from the different sensors. We observed a significant skew in the number of events received at the three hubs in case of certain sensors, e.g., differences of 2357 events (in case of Door 1), 58 events (in case of Motion 1), and 21 events (in case of Motion 3); this skew is due to both radio interference and obstructions (e.g., walls, objects) commonly occurring in homes. Moreover, this problem will be further exacerbated for wearable sensors, where a sensor may be in the vicinity of different processes at different times due to user mobility.

2.2 Motivation

Given the limitations of home networks and the challenges in reliably communicating with sensors, we wanted to understand the impact of event loss on applications. To do so, we surveyed a number of smart-home applications.

For many sensors, an event conveys a specific physical observation to an application, and failing to deliver that event can have grave consequences. For example, Panic-Button [1] and iFall [62] are elder-care apps that process events from a wearable sensor worn by an elder and notify caregivers if a fall is detected [27, 51]. Other apps process events from moisture [2, 4] and fire [12] sensors to notify homeowners. Intrusion-detection apps process door-open events by taking a picture of the person entering and issuing alerts. In these cases, a *gap* in the event stream is clearly undesirable and potentially catastrophic.

Slip&Fall [1] and Daily Routine [1] are SmartThings [15] apps that use motion, door, and cabinet sensor events to detect if an elder did not return from the bathroom after a specified period of time, or if she is not active by a specified time daily, in which case they notify caregivers. In these cases, gaps in the event stream may lead the app to issue false notifications to the caregivers.

In certain applications, gaps in the event stream can lead to incorrect output to the user, while leaving the app with little means to correct it. For example, EnergyDataAnalytics [61] is an app that uses a full-house power sensor to calculate the total energy cost, taking into account the pricing scheme (e.g., time-of-day) and verifies a user's utility bill. In this case, missing events can lead to incorrect reported costs.

While Gapless delivery guarantee is ideal for smart-home applications, some apps, however, can live without this strong guarantee. For such apps, short-lived gaps do not have a noticeable or catastrophic impact. Table 1 summarizes these applications and their mandate for either Gap or Gapless delivery. For instance, consider an app that uses occupancy sensors to set the target temperature of a thermostat [58]; when missing sensor values, the app uses pre-determined policy or defaults to the last set temperature. Similarly, apps that infer home occupancy (e.g., to automate home lighting [1]), can tolerate short-lived gaps in the event stream of the occupancy sensor by inferring occupancy from other sensors such as door open, microphones, or cameras.

Consequently, a reliable event delivery service is needed, one that runs across available hubs, ingests events from in-range sensors, and delivers events to applications running on different hubs.

3 Rivulet Overview

In this section, we first explain the design assumptions. We then give a high level overview of Rivulet's applications. Finally, we introduce the system deign, and give an overview of Rivulet's deployment and execution of applications. Each instance of Rivulet runtime (process for short hereafter) has IP connectivity and is able to communicate with other processes. These processes run on smart phones, tablets, and home appliances which are increasingly well-equipped with a reasonable degree of computing/memory resources.

3.1 Design Assumptions

In Rivulet, we consider a crash-recovery failure model: a process behaves correctly when running, and halts all activity when it crashes. Failed processes are assumed to recover eventually (or permanently taken out of service).

We assume that communication among processes occurs over a transport layer that provides a reliable, in-order point-to-point message delivery protocol, such as TCP/IP. Further, the inter-process network may suffer from arbitrary partitions, e.g., due to one or more faulty network devices such as a home WiFi router.

We assume that sensors and actuators can crash and recover. A crashed sensor simply reports no events, while a faulty actuator does not respond to commands. Moreover, sensors and actuators have very limited compute power (e.g., micro-controller), and are unable to run Rivulet processes on themselves.

Lastly, we assume a best-effort communication layer between every sensor/actuator and processes: each sensor is able to send sensed events to a *subset* of processes, and each actuator is able to receive events from a *subset* of processes. This allows our model to be applicable to a wide variety of (low-power) wireless technologies that are used by off-the-shelf sensors/actuators, e.g., Zigbee [20], ZWave [19], BLE [7], and TCP/IP. For instance, by leveraging a mesh network, a ZWave sensor is capable of sending its events to all the in-range processes that are capable of talking ZWave, while a BLE sensor may only send its events to a single process running on a host with BLE support.

3.2 Apps as Directed Acyclic Graphs

Rivulet applications are built as *directed acyclic graphs* with three types of nodes: sensor, logic, and actuator. Sensor nodes represent physical sensors and are the source of events, logic nodes encapsulate application-specific processing, and actuator nodes represent physical devices for the app to control. Event streams flow from sensor nodes to logic nodes, and command streams flow from logic nodes to actuator nodes.

For instance, consider an application that (i) turns on a light (called LightActuator) whenever a door sensor (called DoorSensor) emits a door-open event, and (ii) turns the light off whenever the door sensor emits a door-close event; the developer constructs the following graph:

$$\text{DoorSensor} \Rightarrow \text{TurnLightOnOff} \Rightarrow \text{LightActuator}$$

The DoorSensor node encapsulates the logic of receiving data values (called events) from the physical sensor, the LightActuator transmits actuation commands to the light switch, while the TurnLightOnOff (TL) node checks if the door has been opened (or closed) and turns the light on (or off) accordingly. Since most smart-home applications are stateless [1], Rivulet does not natively support stateful applications; applications are free to use existing distributed storage systems to replicate state.

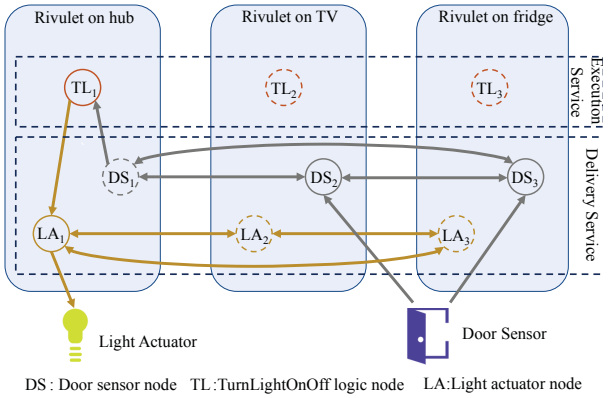


Figure 2. Rivulet System

3.3 System Design

We now explain how Rivulet deploys and executes our example application graph in a scenario with three hosts: a TV, a fridge, and a hub (Figure 2). Each host runs an instance of a Rivulet process providing delivery and execution services.

For each sensor or actuator node in the graph, each process creates either an *active node* denoted by solid circles, or a *shadow node* denoted by dashed circles in Figure 2. An active sensor node (or active actuator node) is created if and only if a process can directly communicate with a sensor (or actuator). A process can directly communicate with a sensor (or actuator), if its host has the necessary hardware to directly communicate with the sensor (or actuator), and the sensor (or actuator) is within its range. Otherwise, the process creates a shadow node.

For instance, assume that only the TV and the fridge can directly talk to the door sensor. Hence, they both create active nodes for the door sensor (i.e., DS_2 and DS_3). The hub, on the other hand, is unable to directly communicate to the door sensor, and creates a shadow node DS_1 representing the door sensor. Similarly, the light actuator can solely talk to the hub, therefore creates active node LA_1, and the TV and fridge create shadow nodes LA_2 and LA_3.

A sensor shadow node can receive events from its active peers in other processes and forward them to its peers or to local logic nodes. An actuator shadow node can receive actuation commands and forward them to its active peers in other processes. These shadow nodes thus acts as placeholders, giving logic nodes the illusion that all sensors/actuators are available locally; this enables a simple application programming model (Section 6).

For each logic node in the application graph, every process either creates an *active node* or a *shadow node*. Active logic nodes receive input events, perform required computations using them, and emit the result to local actuator nodes. Shadow logic nodes, on the other hands, are simply placeholders, and provide no specific functionality. In Figure 2, the

hub has an active logic node TL_1 which receives events forwarded by the shadow sensor node DS_1, and emits resulting commands to the active actuator node LA_1.

Rivulet's delivery service is responsible for forwarding and delivering events between nodes based on the configured Gap and Gapless delivery guarantees (Section 4). Rivulet's execution service employs an active-passive replication technique for executing logic nodes (Section 5). Application logic nodes, e.g., TurnLightOnOff, are instantiated as active nodes on one process (e.g., hub) and as shadow nodes on other processes (e.g., TV and fridge). To simplify the discussion in the paper we assume that an application program is encapsulated into a single logic node.

4 Delivery Service

In this section, we describe Rivulet's delivery service and its provision for Gap and Gapless delivery guarantees for both push-based and poll-based sensors. Push-based sensors detect, or respond to, physical phenomenon by emitting "events"; for example, a "door open" or a "motion" event. Poll-based sensors, on the other hand, generate events only in response to requests; for example, a temperature or a humidity sensor is polled for the current value. For ease of discussion, we assume that the time length of the polling epoch is defined such that the app requires one event per epoch; for example, one temperature event every 10 seconds [36]. The delivery service comprises of two components: *(1) Event ingest* fetches events from sensors, and delivers commands to actuators. Since push-based sensors pro-actively send events to processes, the delivery service has little role to play. For poll-based sensors, however, Rivulet coordinates across processes to perform more efficient polling. *(2) Event forwarding* for reliable delivery of events to active logic nodes for subsequent execution. We limit the discussion focus on delivering events from sensors to applications as the delivery of actuation commands is analogous.

4.1 Gapless Delivery

The goal of Gapless delivery is to ensure that any event received from a sensor by any correct process will be eventually delivered to, and processed by, the applications that are interested in that event. Rivulet employs a coordinated polling mechanism along with a novel ring based protocol to ingest and forward events in order to provide Gapless.

Note that Rivulet provides the Gapless guarantee *post ingest*. In other words, if an event never gets delivered to any of the processes from a push-based sensor, there is no way that Rivulet can notice this, and hence the system cannot provide any guarantee. On the other hand, for poll-based sensors, Rivulet can detect a lack of event delivery in an epoch, and can notify the application by throwing an exception.

Event Ingest. For push-based sensors, events arrive at processes pro-actively; since Rivulet has no control over event arrivals, it cannot employ any special mechanisms for these types of sensors. For poll-based sensors, multiple processes can potentially poll without coordination leading to increased contention and battery drain on the sensors; Rivulet mediates to improve poll efficiency.

The simplest solution to provide Gapless delivery for poll-based sensors is to allow all active sensor nodes to periodically poll the physical sensor without any coordination. Note that once processes receive events from sensors, they can employ event forwarding across the ring. This approach, however, will lead to over-polling leading to increased battery drain on the sensor.

Moreover, we observed (Figure 8 in Section 8.5) that many off-the-shelf sensors only support one outstanding poll request, and simply drop the other requests, often silently leading to undesirable semantics. Consequently, an uncoordinated approach will also substantially increase the number of failed poll requests in cases where sensors do not handle concurrent requests; this leads to an adverse impact on application behavior by introducing delays, time outs, and even triggering unexpected application code paths.

To address the above issues, the delivery service employs coordination when polling. Upon initialization, active sensor nodes select their polling schedules such that no concurrent requests are issued to the sensor. We note that sensor nodes do not need to communicate with each other to select their polling schedules. For example, sensor node i can start polling at time $(i * e)/n$ of every epoch where n is the number of active sensor nodes, and e is the epoch duration. This is possible because applications' epoch lengths are typically significantly larger than the time taken to poll a sensor, e.g., 10 seconds epochs as compared to a 500 ms polling period in case of a ZWave temperature sensor.

Active sensor nodes then proceed to poll the physical sensor, and broadcast the received event using event forwarding service explained next. If an active sensor node receives an event for the current epoch via event forwarding from another node, it simply cancels its scheduled poll for that epoch. Thus, in the more common failure free case, and as long as the time taken to propagate an event across nodes is smaller than the epoch duration, a sensor is only polled once per epoch. We evaluate the benefit of coordinated polling using real sensors in Section 8.5.

Event Forwarding. Since the process that hosts an application may fail in the middle of processing an event, the event forwarding component needs to replicate ingested events at all available processes along with delivering them to applications; this is key to the fault tolerance provided by Rivulet. Replicating events at all available processes guarantees that as long as one correct process exists in the system, that process eventually promotes all its shadow logic nodes to active logic nodes; this consequently ensures that

all outstanding events are delivered to active logic nodes for processing (Section 5).

In order to replicate an event at all available processes, one approach would be to broadcast received events (at different processes) to all other processes using a well-known (reliable) broadcast protocol [23, 63]. However, conventional broadcast protocols designed for a server environment impose high overhead for the common failure-free scenario; sensor-process link losses are relatively rare in many cases (e.g., 0.01% in Figure 1). This is because in the absence of such link loss, an event from a multicast-based sensor will be received at multiple processes, and broadcasting it from every receiving process would impose a significant overhead on all processes and the home network. Hence, Rivulet adopts a more optimistic approach wherein a process first tries to propagate a received event using a novel light-weight *ring* protocol, which we describe next; only if it fails, does Rivulet resort back to reliable broadcast [23].

Every process p_i maintains a local view (denoted as v_i) of potentially available processes. Observe that p_i always exists in v_i since process p_i never suspects itself. The process local view is generated by exchanging keep-alive messages every t seconds with other processes. Since Rivulet must work with any number of processes, including home environments with only one or two processes, majority-based distributed protocols for maintaining agreed-upon views cannot be used. Thus, local views of different processes may be inconsistent.

The protocol uses the following message format: $(e : S : V)$ where e denotes the event, S denotes the list of processes that have seen the event, and V denotes the list of processes that need to see the event. If an active sensor node at process p_i receives an event e from the physical sensor and this event was not previously seen, it sends message $(e : \{p_i\} : v_i)$ to its ring successor sensor node according to its local view v_i. By incorporating its local view v_i, process p_i declares the list of processes that suppose to deliver the event. Additionally, if the process also contains an active logic node, it delivers the event to the logic node so event can be processed.

Upon receiving $(e : S : V)$ from another process, the sensor node at process p_i first checks whether it has previously seen the event or not. If the event has not been seen, it sends message $(e : \{p_i\} \cup S : v_i \cup V)$ to its immediate successor based on its local view v_i. Otherwise, the sensor node evaluates the following two conditions:

- $S \neq V$, which holds true if due to some failure (or asynchrony), certain processes did not receive the event.
- $p_i \in S$, which holds true if process p_i previously had seen the event.

If both conditions hold true, p_i knows that: (1) it has previously forwarded the event to its successor, and (2) the event has not been delivered to all processes (as per the local view of some process). Consequently, process p_i initiates a reliable broadcast to send the event to all available processes. Otherwise, the received event is ignored.

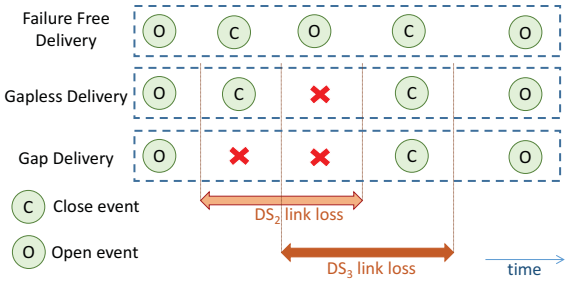


Figure 3. Gap and Gapless Deliveries Under Failures

Whenever process p_i updates its local view v_i and has a new successor, it synchronizes its set of received events with the new successor, and re-sends every event that the new successor has not received. To perform this synchronization efficiently, p_i first queries the new successor for the timestamp of the last event it has received, as in Bayou [68]. It then computes the set of events that need to be sent to the new successor.

Figure 3 shows an example of Gapless delivery in case of failures. Observe that due to concurrent link losses, the third event never reaches any of the processes. Therefore, Rivulet cannot do anything about it. But unlike Gap delivery, the second event will be delivered despite the link loss.

In addition to working for any number of processes, our Gapless delivery protocol improves the network overhead substantially. It benefits from the direct receipt of an event at multiple (say m) processes, and only requires n messages where n is the number of processes. In contrast, using a broadcast protocol that initiates broadcasts from every process that receives an event from the physical sensor incurs a messaging overhead of $O(m \times n)$ even in the failure free case [23].

4.2 Gap Protocol

Gap delivery provides best-effort delivery of events from sensors to logic nodes and actuation commands from logic nodes to actuators. However, delivery is not guaranteed in case of failures, such as a process crash, sensor-process link loss, or a network partition. For Gap, Rivulet organizes all sensor nodes for a given sensor, one per process, into a single logical chain.

Event Ingest. Similar to Gapless, the delivery service does not use any special mechanisms for push-based sensors. For poll-based sensors, it uses a simple mechanism to improve polling efficiency. The active sensor node closest to an active logic node, in the chain, periodically polls the sensor. All other active sensor nodes do not poll the sensor. Observe that in case of a failure of the process that is responsible for polling, polling will resume after the next active sensor node in line removes the failed sensor node from its chain, and consequently becomes closest to the active logic node.

Event Forwarding. A sensor node delivers an event that it receives to its co-resident logic node only if this logic node is active. Otherwise, the active sensor node that is closest, in a chain, to the active logic node forwards the event to the sensor node that is co-resident with the active logic node. The receiving sensor node then delivers the event to its co-resident active logic node. Other active sensor nodes that may have received the event simply discard it. For example, consider the example depicted in Figure 2, and assume that the chain order is hub, TV and fridge. DS_3 drops all the events it receives from the door sensor, while DS_2 forwards the received events to DS_1. Since DS_1 has an active logic node, it delivers the events that it receives to TL_1.

Observe that in this protocol, only one process's sensor node forwards the events being received. Consequently, other processes can use their resources for running other tasks. This low-overhead protocol, however, has some drawbacks. Events may not be received by the closest active sensor node in the case of intermittent sensor-process link failures, which could arise from radio interference in Zigbee and Z-Wave sensors. Since Gap delivery makes no extra effort to retrieve an event from other processes that received the event, the event stream delivered to the application may be incomplete (i.e., with gaps).

Figure 3 shows an example of Gap delivery in case of sensor-process link losses. Since the node DS_2 is responsible for delivering the events emitted from the door sensor to the logic node, link-loss failures leads to gaps in event delivery. Thus, the second and third events (Close and Open) are not delivered to the active logic node. Moreover, if a process crashes, some events may never get delivered until the process failure is detected by other processes, and a new sensor node in another process becomes responsible for forwarding events to the logic node.

5 Fault-tolerant Execution

Rivulet uses a simple *primary-secondary* approach for fault-tolerant execution of applications. To this end, it employs a variant of the bully-based leader election algorithm [33] for selecting the active logic node. Whenever a shadow logic node suspects that all its successors in the chain have crashed, it promotes itself to become the active logic node and notifies all its predecessors in the chain. Similarly, whenever an active logic node detects that its immediate chain successor (if any) has recovered, it demotes itself to a shadow node, and notifies all its predecessors in the chain [33].

Devices in a home are often connected to a single WiFi router whose failure can lead to all processes being partitioned from each other. In this case, all shadow logic nodes will promote themselves to active and thus process any event that they receive from their co-resident sensor nodes. The behavior of the execution environment is determined by whether the actuator is *idempotent* or not.

When actuations are idempotent, it is acceptable for multiple instances of the application to run on different processes. Some common examples are turning a light on, setting the HVAC temperature, or locking a door which can be issued multiple times without having an adverse effect. Most of the actuators studied by us, e.g., bulbs, switches, sirens, thermostats, and locks, fall in the idempotent category.

For non-idempotent actuations, more care is needed to prevent unwarranted action. For instance, dispensing water to a plant through a smart water dispenser or asking the coffee maker to brew are non-idempotent. In these scenarios, multiple instances of logic nodes can run concurrently on different processes only if the actuator supports *Test&Set* commands. Logic nodes can query the state of the actuator before atomically setting a new state thus preventing duplicate actuations.

Network partitions and process failures can also lead to scenarios where events from all required sensors are not available to an active logic node (in addition to a sensor failure scenario). In the next section, we explain how Rivulet's programming model addresses this issue.

6 Programming Model

Like many smart-home platforms [15, 35, 60, 64], Rivulet also employs a dataflow programming model. In particular, our programming model is similar to Flink [6], and provides the following features: (i) a programmer does not need to issue explicit read requests to read sensor events. Rivulet decides when and how frequently to poll sensors; (ii) a programmer specifies an upper bound on the event staleness that the application can tolerate, and Rivulet ensures this bound; and (iii) a programmer specifies how events from multiple sensors are to be aggregated along with fault-tolerant assumptions, and Rivulet guarantees the aggregation as long as the assumptions are met.

6.1 Windows and Operators

A logic node internally comprises of a set of operators that are connected as a directed acyclic graph, and process *windows* of values; a window is a contiguous and finite portion of an event stream, with the following properties:

1. A *bounded event buffer* where the bound can be specified in terms of the number of events or the time-span of the events contained.
2. A *trigger policy* that defines when an operator should be presented with the event buffer for consumption. For instance, a programmer can request for events from all door sensors every t seconds (i.e., time window), or whenever n events become available (i.e., count window).
3. An *evictor policy* that defines the purging of events from the event buffer. For example, a programmer can state a policy to remove events older than s seconds, or to only keep the last n events in the window. In addition, she can choose

Window	
`TimeWindow(Time-span, [TriggerPolicy], [EvictorPolicy])`	Initializes a Time Window with the given timespan and optional trigger and evictor policies
`CountWindow(Count, [TriggerPolicy], [EvictorPolicy])`	Initializes a Count Window with the given count and optional trigger and evictor policies

Operator	
`Operator(Name, [Combiner])`	Initializes an operator with a name and optional Combiner
`addUpstreamOperator(Operator, Window)`	Connects the operator to the given upstream operator
`addSensor(Sensor, GAP|GAPLESS, Window, [PollingPolicy])`	Connects the operator to an upstream sensor with the provided delivery guarantee and optional polling policy
`addActuator(Actuator, GAP|GAPLESS)`	Connects the operator to a downstream actuator with the provided delivery guarantee
`handleTriggeredWindow(Window)`	Callback to handle a triggered window.
`emitWindow(Window, Operators[], Actuators[])`	Emits the outcome to downstream operators, and actuators

Table 2. Operator and Window API

```
1  int n=Rivulet.getSensors("door").size();
2  Operator intruder=new Operator("Intrusion", new FTCombiner(n-1));
3  for (Sensor s: Rivulet.getSensorsWithName("door"))
4    intruder.addSensor(s,GAPLESS, new CountWindow(1)); ...
```

Listing 1. Intrusion Detection

whether to clear the buffer upon a successful trigger or not. Observe that the former case leads to disjoint batches while the latter one can be used for implementing sliding windows. Table 2 summarizes Rivulet's operator and window API.

Different window semantics can be realized for an input stream by combining different buffer types, triggers, and eviction policies. For example, an HVAC-control application computing the average temperature every 60 seconds can use a time window of 60 seconds. Processing events from sensors which emit bursts of events can be simplified using a count window of size 3 for a burst size of 3, so that the operator can easily suppress duplicate-value events. Finally, a home-surveillance application [29] computing the median of last N images' pixels [54] to estimate the background, can use the sliding count window to this end. Programmers can then implement any arbitrary logic to handle triggered windows through the provided callback.

Rivulet allows programmers to specify how triggered windows from different input streams get combined together before being delivered to the operator. To this end, programmers need to implement an interface called Combiner, and pass an instance of it to an operator. Rivulet also provides a specific implementation of combiner interface called FTCombiner that allows applications to easily specify their fault tolerance assumptions, and remains available in case some input streams from some sensors become unavailable. To this end, and for every operator, a programmer only needs to specify the number of sensor failures that it can tolerate. The FTCombiner then delivers triggered windows as long as failure assumptions hold true. We illustrate this further through two example applications.

6.2 Example Applications

First, consider an *intrusion detection* app setting the siren on a door open. Listing 1 shows how a programmer wires an operator to sensors, and specifies its window and FTCombiner. The intruder operator uses count windows of size 1 for its input stream. The programmer also declares that the intruder logic can tolerate up to $n-1$ sensor failures. Therefore, as soon as an event arrives from a door sensor, it is delivered to the intruder operator. Observe that the programmer also configures Gapless delivery for door sensors due to the needs of intrusion detection.

Second, consider a *temperature monitoring* app reporting the "average" home temperature using a number of temperature sensors deployed in the home; averaging of multiple sensor values is fairly common across a variety of apps and hence a useful operation. Marzullo [50] introduced the following algorithm to compute an average of n interval values when at most f sensors can fail: the average value is $[l, u]$ where l is the smallest value in $n-f$ of interval values, and u is the largest value in at least $n-f$ interval values. Depending on the failure model, the above algorithm can tolerate different number of failures. For instance, in order to solely tolerate fail-stop sensors, f needs to be at most $n-1$. On the other hand, to tolerate arbitrary failures, f needs to be at most $\lfloor (n-1)/3 \rfloor$ [50]. Listing 2 shows the wiring of temperature sensors for taking an average every 1 second while tolerating arbitrary failures. Note that by simply passing $n-1$ instead of $\lfloor (n-1)/3 \rfloor$, the averaging operator can only tolerate fail-stop sensors. Finally, we note that to build a complete temperature monitoring app application, one can add the above `averageTemp` operator as its source.

```
int n=Rivulet.getSensors("temperature").size();
Operator averageTemp=new
  Operator("Averaging", new FTCombiner(Math.floor((n-1)/3)));
for (Sensor s: Rivulet.getSensorsWithName("temperature"))
  averageTemp.addSensor(s,GAP, new TimeWindow(1000)); ...
```

Listing 2. Temperature Sensor Averaging

7 Implementation

Our Rivulet prototype is implemented in Java, as a cross-platform service that can be deployed on Linux, Windows, and Android, with around 8k SLOC. On Android, Rivulet runs as a user-application using a single background service task. Rivulet uses the Netty library [16] to manage TCP connections between processes, along with custom serialization for events and other messages. Similar to HomeOS [30] and Beam [60], adapters in Rivulet encapsulate communication specific logic. Rivulet currently implements adapters for Z-Wave, Zigbee, IP cameras [10], and smartphone-based sensors such as microphone, accelerometer, and GPS (on Android). Adapters deliver received events to local active sensor nodes. They also receive actuation commands from local actuator nodes, and translate them to technology-specific network frames, depending upon the type of the adapter. At initialization, Rivulet instantiates the adapters depending upon the communication capabilities of the local process.

The Z-Wave adapter uses a version of the OpenZWave [13], modified by us to enable concurrent receipt of events from multiple sensors and to enable concurrent polling of multiple sensors from a single process. The Zigbee adapter uses the EmberZNet library [8], whereas the IP camera adapter and the smartphone adapters leverage the REST-interface and Android Sensor Manager interface respectively.

The current implementation uses a simple deterministic function to order and select processes for deploying active logic nodes which seeks to deploy a logic node on a process that has the largest number of active sensors and actuators required by the logic node; this allows Rivulet to minimize delay incurred during event delivery.

8 Evaluation

In this section, we answer the following key questions: (i) What is the overhead of Gap and Gapless delivery guarantees, in failure free scenarios? (ii) How effective is Rivulet in handling sensor-process link losses and process failures? and (iii) What is the benefit of coordinated polling mechanism?

8.1 Setup

We evaluated Rivulet in a sample home scenario. We chose Raspberry Pi (Model 3), each with a 1.2 GHz 32-bit quad-core ARM Cortex-A53 processor, 1 GB RAM and Broadcom BCM43438 WiFi card, as Rivulet hosts. Such hardware configuration is representative of the computational capabilities of in-home compute devices which are increasingly equipped with modest computing/memory resources capable of running multiple applications, e.g., smart TVs [14], smart fridges [9], and SmartThing's hubs [15].

Type	Event Size	Examples
Small	4 - 8 B	Temperature, humidity, motion, moisture, door-window open/close, UV level, energy, vibration sensors [3, 13, 20]
Large	1-20 KB	IP camera [5, 10], microphone [42]

Table 3. Classification of off-the-shelf sensors.

In order to emulate a typical home (with a TV, fridge, washing machine, hub, and personal assistant), we spread the hosts across a sample home, and connected them using a single 2.4 GHz IEEE 802.11 a/b/g/n WiFi router. Each experiment run spanned 200 seconds. Reported results are averaged across at least 10 runs.

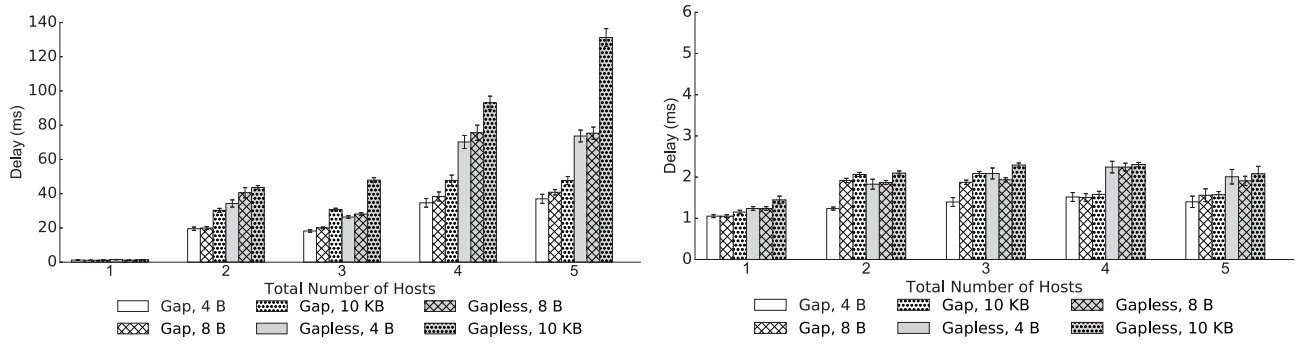
To study different variations across home scenarios, we implemented an IP-based software sensor. This allowed us to: (i) control which processes can or cannot receive events from a given sensor, e.g., due to the communication capabilities and physical topologies, (ii) control the levels of sensor-process link losses, and (iii) remove any clock-skew between sensors and the active logic node.

We surveyed a range of off-the-shelf sensors compatible with various smart-home platforms (e.g., Smartthings [15] and Wink [18]), and found that they can be classified into two broad categories called small type and large type sensors, based on the size of the events they emit, as shown in Table 3. Most sensors measuring physical phenomena such as temperature, humidity, and motion, use a small event size of 4 to 8 bytes, and have a maximum event frequency in the 1 to 10 events per second range. The home IP cameras have small resolutions and used compressed image formats (e.g., JPEG) causing image event sizes in the 10 to 20 KB range with a frame rate of up to 10 frames per second. Our Rivulet prototype supports video streams by discretization into image event streams.

Microphone samples typically use 2 to 3 bytes per sample. However, microphone applications usually consume large batches of samples as a single event. For example, 512 samples per frame (sampled at 8 kHz) for an activity tracking application [42] leads to an event size of 1 KB.

8.2 Gap vs. Gapless Under No Failures

To quantify and compare the overhead of Rivulet's Gap and Gapless guarantees, we measure two key metrics: (i) *delay*: the difference between the time an event is emitted by a



(a) *Worst* topology. **(b)** *Best* topology.

Figure 4. Delay incurred with increasing number of processes, for different event sizes.

sensor and the time it is received by an active logic node, and (ii) *network overhead*: the amount of data transferred over the home network for delivering an event. Ideally, any middleware such as Rivulet should incur minimal delay for allowing home applications to respond to changes in the physical phenomenon in a timely manner. Similarly, the network overhead should also be minimal because home networks are typically shared by a range of user applications, and should be least impacted.

Delay. To study Rivulet's event delay, we consider a sample scenario with one sensor and configure it such that only a single process is able to receive its events. We then increase the total number of Rivulet processes, and vary the placement of the event-receiving process relative to the *application-bearing process* (i.e., the process where the active logic node is placed). The sensor's event rate is fixed at 10 events per second. We measured the delay for the two different delivery guarantees and different event sizes (Table 3).

Figure 4a shows the delay incurred with increasing number of processes, when the process receiving the events is placed farthest from the application-bearing process. We observe that, in case of Gap, for a given event size, the incurred delay increases slightly with increasing number of processes, due to increasing keep-alive message exchange.

In case of Gapless, the delay incurred remains largely unchanged betwen 2 to 3 processes, and increases linearly from 3 to 5 processes. This is because Gapless delivery uses a ring topology to forward the event to all processes, whereas a Gap delivery simply forwards it from the receiving process to the application-bearing process. However, with small number of processes, Gapless delivery incurs only a small additional delay as compared to Gap: 8-10 ms at 2-3 processes for 4B and 8B event sizes, which in our experience constitute a majority of home deployments in operation today.

Lastly, observe that for a given a number of processes, the delay increases directly with increasing event size. This is attributed to increased network transfer and serialization/deserialization for larger events.

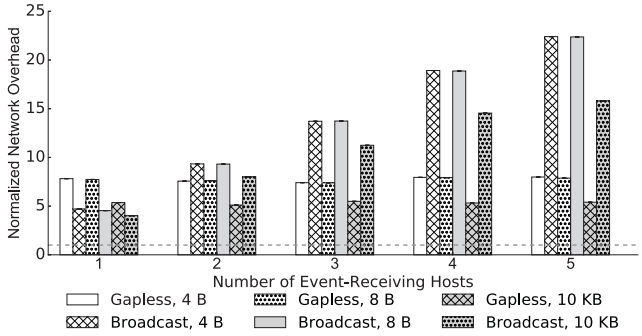


Figure 5. Network overhead normalized against Gap, with increasing number of even sizes and receiving processes.

Figure 4b shows the delay incurred with increasing number of processes, when the application-bearing process is able to directly receive events from the given sensor. In this case, the delay incurred is relatively low and is approximately in the 1 to 2 ms range.

Network Overhead. Figure 5 shows the network overhead in case of Gapless, normalized against that of Gap (dotted line). The total number of processes is fixed to five and the number of event-receiving processes is varied from one to five. Figure 5 also shows the normalized network overhead of a simple broadcast approach in which a process broadcasts an event to other processes upon receiving the event from the sensor, unless it has previously received the event from another process (as explained in Section 4.1).

We observe that our Gapless delivery protocol incurs a constant network overhead (albeit higher than Gap), regardless of the number of event-receiving processes. In contrast, a broadcast-based approach incurs a 23% higher overhead in case of 2 event-receiving processes. This overhead increases further to 2× higher with 3 processes, and 3× higher with 5 event-receiving processes (for event size of 4 bytes). However, due to the metadata transferred by the Gapless delivery protocol (sets S and V), it has a higher overhead at 1 event-receiving process. Also observe that the normalized

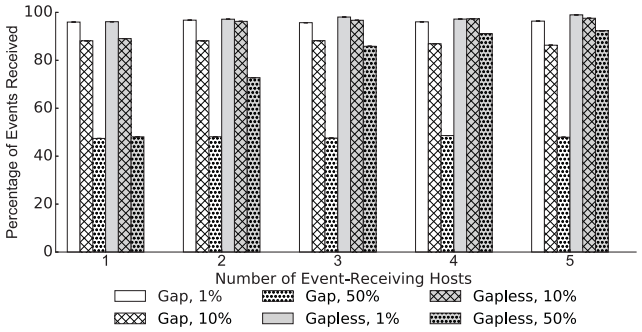


Figure 6. Percentage of events delivered with increasing number of event-receiving processes.

network overhead is lower at large event size than at small event sizes. This is because large event sizes amortize the network overhead of any metadata, e.g., message headers, that is transferred by a protocol in addition to the event.

8.3 Sensor-process Link Loss

To study the benefit of Rivulet's Gapless delivery in the presence of link loss, we consider a sample scenario with one sensor. We increase the number of processes that are able to receive events from this sensor. In addition, we induce loss of events on each sensor-process link at different link loss rates. The total number of processes is fixed at five, and the event-receiving processes are placed farthest from the application-bearing process.

Figure 6 shows the percentage of events received by an application in this scenario, with increasing link loss rate and increasing number of event-receiving processes. The event size is 4 bytes and the event rate is 10 events per second. We observe that at low link loss rates, Gap delivers approximately the same number of events as Gapless regardless of the number of event-receiving processes. However, as the link loss increases, Gapless delivery is able to retrieve events across multiple processes and delivers them to the active logic node. For instance, at 10% link loss and with 2 event-receiving processes, Gap delivers 90% of the emitted events because it forwards events from a single receiving process. In contrast, Gapless delivers 99% of emitted events, i.e., the percentage of events received in *at least* one process

At 50% link loss, Gap delivers only 50% of the emitted events, whereas Gapless delivers approximately 75%, 87%, and 95% with two, four, and five receiving processes respectively. We observe a similar trend for other event rates and sizes, thus we omit their analysis due to space limitations.

8.4 Process Failure

To illustrate Rivulet's tolerance of process failures, we induce a process failure in a sample scenario with one sensor generating 10 events per second. The total number of processes and the number of event-receiving is set to five.

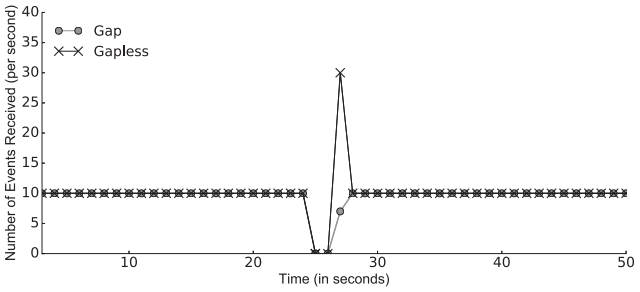


Figure 7. Number of events received by an active logic node. Induced process failure at $t = 24$ seconds.

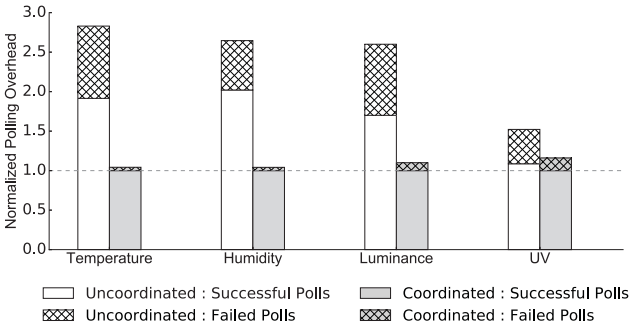


Figure 8. Normalized polling overhead for sensors.

Figure 7 shows the number of events received at the application over time, during a single such experiment run. We crash the initial application-bearing process after 24 seconds. Rivulet then detects the failure, selects a new process as primary, and promotes the shadow logic node on it to an active logic node. In case of Gapless delivery, the logic node receives the events that were emitted (or were in-flight) while handling the failure. However, in case of Gap, the logic node simply receives the next available event. The failure-detection time threshold in Rivulet is set to two seconds, thus leading to a gap of approximately 20 events in case of Gap delivery. In case of Gapless, this causes a spike in the number of received events at $t = 27$ seconds, due to the 20 additional events.

8.5 Coordinated Polling

To illustrate the benefit of Gapless delivery service for poll-based sensors, we consider a sample scenario with three processes, and four poll-based sensors. We use Z-Wave temperature, luminance, relative humidity, and ultraviolet (UV) radiation sensors [19], which have a polling period of 600 ms, 600 ms, 4 seconds, and 5 seconds respectively. Our sample application requests epochs of length 1800 ms, 1800 ms, 12 seconds, and 15 seconds respectively for these sensors.

Figure 8 shows the number of poll requests issued to a sensor (in coordinated and uncoordinated way) normalized against the number of polls required in the optimal case, i.e.,

once per epoch (shown as a dotted line). In the uncoordinated case, each process issues one poll request uniformly randomly within each epoch. The coordinated approach, on the other hand, uses the simple mechanism introduced in Section 4.1.

Our coordinated polling mechanism incurs 4 to 13% additional requests as compared to the optimal case. This is due to (i) delays during ring propagation of an event that causes redundant polling for the same epoch, and (ii) failed poll requests requiring re-polling. In contrast, uncoordinated polling causes significantly higher poll requests than optimal, ranging from 1.5× to 2.5× higher. This approach therefore can lead to 1.5 to 2.5 × lower sensor battery life, while Gapless delivery with coordinated approach incurs a 4 to 13% decrease. Observe that Gap delivery incurs the optimal polling overhead but may fail to generate an event for every epoch in case of failures.

9 Related Work

Platforms for Smart Homes. Semantic Streams [70] and Task Cruncher [67] are frameworks that allow applications to compose sensor streams into a DAG. However, their model with a central server has a single point of failure. Beam [60] and HomeOS [30] are frameworks to ease smart-home application development, similar to Rivulet. Beam partitions applications across devices to optimize the utilization of one or more resources including CPU, memory, network bandwidth, or battery. Similarly, Rover [41] provides a programming framework that eases developed by providing abstractions to partition mobile applications. MagnetOS [47] partitions a set of communicating Java objects in a sensor network while addressing energy efficiency. This body of work does not address tolerating failures that occur regularly in home environments such as process failures, network partitions, sensor-process link losses. However, Rivulet can benefit from their optimization techniques, for instance, to perform dynamic placement of nodes on processes.

Stream processing frameworks. There exists a large body of work on fault-tolerant stream processing frameworks [21, 22, 25, 26, 40, 45, 49, 59, 69], which introduces a variety of fault-tolerance mechanisms ranging from upstream backups, checkpointing, tentative tuples, to active replication for tolerating process and network failures. However, one central assumption made in this body of work is that *ingress* data is always available at all processes. As described in Section 4 and Section 4.1, stabilization of ingress data in a home, due to sensor-process link losses, poses an additional challenge. As we demonstrate, given the wimpy nature of in-home compute devices, a simple active-passive replication can suffice.

Hwang et al. [40] introduced three recovery types for stream processing systems – *precise* that the output of an execution with failures is identical to a failure-free execution, *recovery* ensures no information loss, and *gap* may cause information loss – of which, recovery and gap are conceptually similar to Rivulet's Gap and Gapless. However, Rivulet specializes these guarantees for smart-home scenarios comprising sensors and processes, and provides novel low-cost mechanisms for implementation.

Martin et al. [49] present a comprehensive fault-tolerant processing system that uses a combination of techniques, including passive and active replications, to tolerate failures. Although Rivulet's active-passive approach suffices for most current applications, other approaches introduced by Martin et al. [49] can be leveraged for reducing recovery times.

Sensor failure tolerance. Recent work [44, 52] has focused on detecting and tolerating sensor failures by correlating events from different sensors, e.g., tolerating a failed door-sensor using co-located motion sensors. Other work [28, 31, 50, 53] has proposed generalized techniques to tolerate arbitrary sensor failures. As described in Section 6, these algorithms are complimentary to Rivulet, can be programmed in Rivulet as nodes, and motivated Rivulet's programming abstractions.

Wireless Sensor Networks (WSN). Rivulet was substantively inspired by work in WSNs [37, 38, 48, 55, 65] but differs from them in the following ways. First, existing work focuses primarily on sensor data collection and data routing, whereas Rivulet is a platform for building and executing smart-home applications operating on sensor events. Second, existing work on reliable WSNs focuses on experimental sensor deployments using a single network technology (e.g., Zigbee), whereas Rivulet supports multiple networks. Third, existing WSNs mainly operate at the network layer while Rivulet acts as a middleware between the network layer and applications, abstracting the complexities in building fault-tolerant smart-home applications.

10 Conclusion

Rivulet is a distributed "smart-home" platform that provides fault-tolerant delivery of sensor events and actuation commands to enable a new class of robust smart-home applications. Rivulet is designed to be pragmatic about the failures and recovery options available in a home in contrast to a managed environment such as a data center. Rivulet does not rely on a majority of non-faulty nodes, and operates with a diverse set of low-power wireless networks. Our evaluation shows that Rivulet meets the needs of common smart-home applications and provides low-overhead fault tolerance.

Acknowledgments

We would like to thank Juan Colmenares, Tanakorn Leesatapornwongsa, Iqbal Mohomed, Aritra Sengupta, Ahmad Tarakji, Ashish Vulimiri, the anonymous reviewers, and our shepherd, Ramya Raghavendra, for their insightful feedback and suggestions.

References

[1] 2015. Smart Things Officially Published Apps. https://community.smartthings.com/t/list-of-all-officially-published-apps-from-the-more-category-of-smart-setup-in-the-mobile-app-deprecated/13673. (2015).

[2] 2015. SmartThings Home Flood App. https://blog.smartthings.com/how-to/how-to-prevent-a-leak-from-causing-a-flood/. (2015).

[3] 2016. Z-Wave Device Class Specification. http://zwavepublic.com/sites/default/files/SDS10242-29%20-%20Z-Wave%20Device%20Class%20Specification.pdf. (2016).

[4] 2017. Aeotec Z-Wave Home Sensors. http://aeotec.com/homeautomation. (2017).

[5] 2017. Amazon Home Automation Store. (2017). https://www.amazon.com/home-automation-smarthome/b?ie=UTF8&node=6563140011.

[6] 2017. Apache Flink. https://flink.apache.org/. (2017).

[7] 2017. Bluetooth Low Energy. https://www.bluetooth.com. (2017).

[8] 2017. EmberZNet PRO ZigBee Protocol Stack Software. https://www.silabs.com/products/development-tools/software/emberznet-pro-zigbee-protocol-stack-software. (2017).

[9] 2017. Family Hub Refrigerator. http://www.samsung.com/us/explore/family-hub-refrigerator/overview/. (2017).

[10] 2017. Foscam IP Camera. http://foscam.us/. (2017).

[11] 2017. HOmeKit. https://developer.apple.com/homekit/. (2017).

[12] 2017. Nest Protect. https://nest.com/smoke-co-alarm/overview/. (2017).

[13] 2017. OpenZWave. https://github.com/openzwave/. (2017).

[14] 2017. Samsung JS9000 SUHD TV. http://www.samsung.com/us/system/consumer/product/un/48/js/un48js9000fxza/TVJS9000SpecSheet3-13-15.pdf. (2017).

[15] 2017. SmartThings. (2017). http://www.smartthings.com/.

[16] 2017. The Netty Project. http://netty.io/. (2017).

[17] 2017. Vera Smarter Home Control. (2017). http://getvera.com/.

[18] 2017. Wink Connected Home Hub. (2017). http://www.wink.com/.

[19] 2017. Z-Wave Alliance. http://www.z-wavealliance.org. (2017).

[20] 2017. ZigBee Alliance. http://www.zigbee.org/. (2017).

[21] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *Proc. VLDB Endow.* 734–746.

[22] Magdalena Balazinska, Hari Balakrishnan, Samuel R. Madden, and Michael Stonebraker. 2008. Fault-tolerance in the borealis distributed stream processing system. In *Int. Conf. on the Mgt. of Data (SIGMOD)*. 13–24.

[23] Romain Boichat and Rachid Guerraoui. 2005. Reliable and total order broadcast in the crash-recovery model. *Journal of Parallel and Dist. Comp.* 65, 4 (April 2005), 397–413.

[24] A. J. Brush, Jaeyeon Jung, Ratul Mahajan, and Frank Martinez. 2013. Digital Neighborhood Watch: Investigating the Sharing of Camera Data Amongst Neighbors. In *Conf. on Computer Supported Cooperative Work (CSCW)*.

[25] Don Carney, U\vgur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2002. Monitoring Streams: A New Class of Data Management Applications. In *Int. Conf. on Very Large Data Bases (VLDB)*. 215–226.

[26] Guoqiang Jerry Chen, Janet L Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime Data Processing at Facebook. In *Int. Conf. on the Mgt. of Data (SIGMOD)*. 1087–1098.

[27] Jay Chen, Karric Kwong, Dennis Chang, Jerry Luk, and Ruzena Bajcsy. 2005. Wearable sensors for reliable fall detection. In *Engineering in Medicine and Biology Society*. 3551–3554.

[28] Paul Chew and K Marzullo. 1991. Masking failures of multidimensional sensors. In *Symp. on Reliable Dist. Sys. (SRDS)*. 32–41.

[29] Chun-Te Chu, Jaeyeon Jung, Zicheng Liu, and Ratul Mahajan. 2014. *sTrack: Secure Tracking in Community Surveillance.* Technical Report. Microsoft Research.

[30] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A.J. Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. 2012. An Operating System for the Home. In *Networked Sys. Design and Implem. (NSDI)*. 337–352.

[31] Hugh Durrant-Whyte and Thomas C Henderson. 2008. *Multisensor Data Fusion.* 585–610.

[32] Peter Xiang Gao and Srinivasan Keshav. 2013. SPOT: a smart personalized office thermal control system. In *Int. Conf. on Future energy systems (e-Energy)*. ACM, 237–246.

[33] Hector Garcia-Molina. 1982. Elections in a Distributed Computing System. *IEEE Trans. on Computers* C-31, 1 (1982), 48–59.

[34] Jessica Groopman and Susan Etlinger. 2015. *Consumer Perceptions of Privacy in The Internet of Things.* Technical Report.

[35] Trinabh Gupta, Rayman Preet Singh, Amar Phanishayee, Jaeyeon Jung, and Ratul Mahajan. 2014. Bolt: Data Management for Connected Homes. In *Networked Sys. Design and Implem. (NSDI)*. 243–256.

[36] Ming He, Wen-Jian Cai, and Shao-Yuan Li. 2005. Multiple fuzzy model-based temperature predictive control for HVAC systems. *Information sciences* 169, 1 (2005), 155–174.

[37] Tian He, Brian M Blum, John a Stankovic, and Tarek Abdelzaher. 2004. AIDA: Adaptive application-independent data aggregation in wireless sensor networks. *ACM Transactions on Embedded Computing Systems (TECS)* 3, 2 (2004), 426.

[38] Tian He, Sudha Krishnamurthy, Liqian Luo, Ting Yan, Lin Gu, Radu Stoleru, Gang Zhou, Qing Cao, Pascal Vicaire, John a Stankovic, Tarek F Abdelzaher, Jonathan Hui, and Bruce Krogh. 2006. VigilNet: An integrated sensor network system for energy-efficient surveillance. *ACM Transactions on Sensor Networks* 2, 1 (2006), 1–38.

[39] Timothy W Hnat, Vijay Srinivasan, Jiakang Lu, Tamim I Sookoor, Raymond Dawson, John Stankovic, and Kamin Whitehouse. 2011. The hitchhiker's guide to successful residential sensing deployments. In *Conf. on Embedded Networked Sensor Sys. (SenSys)*. 232–245.

[40] Jeong Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. 2005. High-availability algorithms for distributed stream processing. In *Int. Conf. on Data Engineering (ICDE)*. 779–790.

[41] Anthony D Joseph, Alan F de Lespinasse, Joshua A Tauber, David K Gifford, and M Frans Kaashoek. 1995. Rover: a toolkit for mobile information access. In *Operating Systems Review*, Vol. 29. 156–171.

[42] Younghyun Ju, Youngki Lee, Jihyun Yu, Chulhong Min, Insik Shin, and Junehwa Song. 2012. SymPhoney: a coordinated sensing flow execution engine for concurrent mobile sensing applications. In *Conf. on Embedded Networked Sensor Sys. (SenSys)*. ACM, 211–224.

[43] Sachin Katti, Shyamnath Gollakota, and Dina Katabi. 2007. Embracing wireless interference: Analog network coding. *ACM SIGCOMM Computer Communication Review* 37, 4 (2007), 397–408.

[44] Palanivel Kodeswaran, Ravi Kokku, Sayandeep Sen, and Mudhakar Srivatsa. 2016. Idea : A System for Efficient Failure Management in Smart IoT Environments. In *Int. Conf. on Mobile Sys., Apps. and Services (MobiSys)*. 43–56.

[45] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Int. Conf. on the Mgt. of Data (SIGMOD)*. 239–250.

[46] Hongbo Liu, Yu Gan, Jie Yang, Simon Sidhom, Yan Wang, Yingying Chen, and Fan Ye. 2012. Push the limit of WiFi based localization for smartphones. In *Int. Conf. on Mobile computing and networking (Mobicom)*. 305–316.

[47] Hongzhou Liu, Tom Roeder, Kevin Walsh, Rimon Barr, and Emin Gün Sirer. 2005. Design and Implementation of a Single System Image

Operating System for Ad Hoc Networks. In *Int. Conf. on Mobile Sys., Apps. and Services (MobiSys)*. 149–162.

[48] Chenyang Lu, B.M. Blum, T.F. Abdelzaher, J.a. Stankovic, and T. He. 2002. RAP: A real-time communication architecture for large-scale wireless sensor networks. In *Real-Time and Embedded Technology and Applications Symp. (RTAS)*. 55–66.

[49] A Martin, T Smaneoto, T Dietze, A Brito, and C Fetzer. 2015. User-Constraint and Self-Adaptive Fault Tolerance for Event Stream Processing Systems. In *Int. Conf. on Dependable Sys. and Networks (DSN)*. 462–473.

[50] Keith Marzullo. 1990. Tolerating Failures of Continuous-valued Sensors. *Trans. on Computer Sys.* 8, 4 (Nov. 1990), 284–304.

[51] Muhammad Mubashir, Ling Shao, and Luke Seed. 2013. A survey on fall detection: Principles and approaches. *Neurocomputing* 100 (2013), 144 – 152. Special issue: Behaviours in video.

[52] Sirajum Munir and John A Stankovic. 2014. FailureSense: Detecting Sensor Failure using Electrical Appliances in the Home. In *Int. Conf. on Mobile Ad Hoc and Sensor Systems (MASS)*. 73–81.

[53] Eduardo F Nakamura, Antonio A F Loureiro, and Alejandro C Frery. 2007. Information Fusion for Wireless Sensor Networks: Methods, Models, and Classifications. 39, 3 (Sept. 2007).

[54] Massimo Piccardi. 2004. Background subtraction techniques: a review. In *Int. Conf. on Systems, Man and Cybernetics*, Vol. 4. 3099–3104.

[55] Joseph Polastre, Jason Hill, and David Culler. 2004. Versatile Low Power Media Access for Wireless Sensor Networks. In *Conf. on Embedded Networked Sensor Sys. (SenSys)*. 95–107.

[56] Shravan Rayanchu, Ashish Patro, and Suman Banerjee. 2011. Airshark: detecting non-WiFi RF devices using commodity WiFi hardware. In *Conf. on Internet Measurement Conference (IMC)*. 137–154.

[57] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing* 8, 4 (2009).

[58] James Scott, A J Bernheim Brush, John Krumm, Brian Meyers, Michael Hazas, Stephen Hodges, and Nicolas Villar. 2011. PreHeat: Controlling Home Heating Using Occupancy Prediction. In *Int. Conf. on Ubiquitous Computing (UbiComp)"*. 281–290.

[59] Mehul A Shah, Joseph M Hellerstein, and Eric Brewer. 2004. Highly Available, Fault-tolerant, Parallel Dataflows. In *Int. Conf. on the Mgt. of Data (SIGMOD)*. 827–838.

[60] Chenguang Shen, Rayman Preet Singh, Amar Phanishayee, Aman Kansal, and Ratul Mahajan. 2016. Beam: Ending Monolithic Applications for Connected Devices. In *Usenix Annual Tech. Conf. (ATC)*. 143–157.

[61] Rayman Preet Singh, S. Keshav, and Tim Brecht. 2013. A Cloud-based Consumer-centric Architecture for Energy Data Analytics. In *Int. Conf. on Future energy systems (e-Energy)*. 63–74.

[62] Frank Sposaro and Gary Tyson. 2009. iFall: an Android application for fall monitoring and response. In *Annual Int. Conf. of the Engineering in Medicine and Biology Society (EMBC)*. 6119–6122.

[63] Ioana Stanoi, Divyakant Agrawal, and Amr El Abbadi. 1998. Using broadcast primitives in replicated databases. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*. 148–155.

[64] Julian James Stephen, Savvas Savvides, Vinaitheerthan Sundaram, Masoud Saeida Ardekani, and Patrick Eugster. 2016. STYX: Stream Processing with Trustworthy Cloud-based Execution. In *Symp. on Cloud Computing (SoCC)*. 348–360.

[65] Vinaitheerthan Sundaram, Saurabh Bagchi, Yung Hsiang Lu, and Zhiyuan Li. 2008. SeNDORComm: An energy-efficient priority-driven communication layer for reliable wireless sensor networks. In *Symp. on Reliable Dist. Sys. (SRDS)*. 23–32.

[66] Jay Taneja, Andrew Krioukov, Stephen Dawson-Haggerty, and David Culler. 2013. Enabling advanced environmental conditioning with a building application stack. In *Int. Green Computing Conference (IGCC)*. 1–10.

[67] Arsalan Tavakoli, Aman Kansal, and Suman Nath. 2010. On-line Sensing Task Optimization for Shared Sensors. In *Int. Conf. on Information Processing in Sensor Networks (IPSN)*. 47–57.

[68] Douglas B. Terry, M. M. Theimer, Karin Petersen, a. J. Demers, M. J. Spreitzer, and C. H. Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Symp. on Op. Sys. Principles (SOSP)*. 172–182.

[69] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm @Twitter. In *Int. Conf. on the Mgt. of Data (SIGMOD)*. 147–156.

[70] Kamin Whitehouse, Feng Zhao, and Jie Liu. 2006. Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data. In *W. on Wireless Sensor Networks*. 5–20.