An Architecture for IoT Clock Synchronization

Sathiya Kumaran Mani[†], Ramakrishnan Durairajan[#], Paul Barford[†], Joel Sommers[*]
[†]University of Wisconsin - Madison [#]University of Oregon [*]Colgate University

Abstract
In this paper, we describe an architecture for clock synchronization in IoT devices that is designed to be scalable, flexibly accommodate diverse hardware, and maintain tight synchronization over a range of operating conditions. We begin by examining clock drift on two standard IoT prototyping platforms. We observe clock drift on the order of seconds over relatively short time periods, as well as poor clock rate stability, each of which make standard synchronization protocols ineffective. To address this problem, we develop a synchronization system, which includes a lightweight client, a new packet exchange protocol called SPoT and a scalable reference server. We evaluate the efficacy of our system over a range of configurations, operating conditions and target platforms. We find that SPoT performs synchronization 22x and 17x more accurately than MQTT and SNTP, respectively, at high noise levels, and maintains a clock accuracy of within ∼15ms at various noise levels. Finally, we report on the scalability of our server implementation through microbenchmark and wide area experiments, which show that our system can scale to support large numbers of clients efficiently.

CCS Concepts
•Networks → Network protocol design; Time synchronization protocols;
Author Keywords
Time; Internet of Things; SNTP; MQTT; Measurement; Wireless

INTRODUCTION
Distributed Internet applications, including gaming, monitoring and real-time control, require that participating hosts have synchronized clocks. Such synchronized clocks on network-connected devices enable shared experiences for users and coordination of application behaviors and interactions at specified times.

Although the issue of clock synchronization in widely-distributed systems has been studied for many years (*e.g.*, see [23, 25, 26]), Internet of Things (IoT) devices introduce several challenges. First, an objective in IoT device design is to minimize costs. This implies the use of lower quality hardware components including oscillators that generate clock signals on those devices. Second, IoT devices are often deployed in environments with a broad range of operating temperatures. Low-quality clocks have been shown to run at widely varying rates depending on temperature [32]. Finally, IoT devices often have limited computation and communication capability, which can constrain their ability to participate in standard clock synchronization protocols.

In this paper we consider the problem of synchronizing clocks in IoT devices with a remote reference source.[1] While Network Time Protocol (NTP) would be a natural solution for clock synchronization, typical configurations require stateful client computation and on-going communication with reference source(s), which make Simple Network Time Protocol (SNTP) and similarly lightweight mechanisms a more attractive choice.[2] The goals of our work are to understand how clocks operate on IoT devices and how they can be synchronized in an accurate and efficient fashion.

The target platforms for our study are the well known Arduino MKR1000 and the Raspberry Pi3, both of which are often used for IoT prototyping. We begin by examining the drift characteristics on these platforms using raw millisecond counters. Our experiments on the different Arduino devices show inaccuracies in synchronization ranging from 700 ms to as high as 1600 ms. Next, we test clock rate stability over a range of temperatures that might be experienced for typical IoT device deployments. We observe clock drift as high as 600 ms over relatively short time periods. Finally, we characterize the stability of the clock hardware by measuring the *Allan variance* [27]. We find that IoT clock hardware shows high variability and less stability than traditional PC clock hardware. These results motivate new synchronization mechanisms that can accommodate lower clock stability and diverse clock drift characteristics.

To improve synchronization of IoT devices, we develop a new clock synchronization architecture that is designed to be scalable, lightweight and to enable synchronization on the order of 10 ms over a range of operating conditions. The central components in our design are *(i)* a lightweight implementation for IoT devices/clients, *(ii)* a scalable implementation of reference servers that compute all key parameters (*e.g.*, clock offset, rate, etc.), and *(iii)* a packet exchange protocol that we call Synchronization Protocol for ioT (SPoT). In our system, IoT clients simply contact a SPoT server and adjust their clocks to the value indicated in SPoT response packets with no additional computation required. This architecture allows SPoT to be used for time synchronization in many IoT deployment scenarios such as environmental sensing and early warning systems, smart homes, smart grids, smart vehicles, factory floor automation, gaming and IoT blockchain applications to name a few.

IoT 2018, October 15-18, 2018, Santa Barbara, CA, USA

 ISBN 978-1-4503-2138-9…$15.00
DOI: http://dx.doi.org/xx.xxxx/xxxxxxx.xxxxxxx

[1] This differs from the problem of synchronizing clocks in a local deployment such as sensor networks, which may not require synchronization with a global reference.

[2] We are not aware of any NTP client implementations for IoT.

The key technical challenge in our work is to develop clock synchronization algorithms that are robust to the wide range of clock drift and offsets that are expected in large IoT deployments. We address this problem by developing two novel algorithms: one that synchronizes clock rates and one that addresses path asymmetry between client and server. The latter is developed in conjunction with a standard filtering method that allows packets with the best estimates to be selected.

We develop prototype implementations of the client and server components of our system to evaluate its efficacy over a range of configurations, operating conditions and target platforms. First, we find that SPoT outperforms MQTT-based clock synchronization mechanism and reports 16x, 19x, and 22x better clock offsets in the presence of three different levels of noise. Similarly, SPoT reported offsets are 3x, 10x, and 17x better than SNTP. Next, we calculate the clock rate error (*i.e.*, Root Mean Squared Error (RMSE) values) using NTP-reported offsets values and find that SPoT consistently maintain accuracy within $\sim$15 ms at different noise levels. Finally, we demonstrate the scalability of our server implementation in a series of micro-benchmark and wide area tests.

BACKGROUND AND ASSUMPTIONS

Clock synchronization errors

Synchronizing a client's local clock with a reference time source consists of calculating two interdependent components: *(i) clock offset*; and *(ii) clock skew* or rate-error, which is the difference in rate or frequency between the client's clock and the remote reference. While the offset synchronization (*i.e.*, calculating offset) is sufficient for general coordination of events among distributed clients, rate synchronization (*i.e.*, calculating clock skew) is necessary to achieve tight synchronization. Surprisingly, while skew is a predominant source of synchronization error [29], it is occasionally overlooked in distributed clock synchronization algorithms. As we discuss below, inexpensive clock hardware is relatively unstable compared to traditional PC clock hardware and significant clock drift between synchronization requests exacerbates error.

Two-way synchronization method

A fundamental operation in distributed clock synchronization is the timestamped exchange of packets between a client and a reference, where exchanges are typically initiated by the client. This method, known as the *two-way exchange*, often begins with the client sending a request packet at time *t1*. The server receives the packet at time *t2* and sends a reply at time *t3*; the reception time of the reply at the client is *t4*. Timestamps *t1* and *t4* are from the client's clock, where as *t2* and *t3* are from the server's clock. The round trip time (RTT) of the exchange is given by $RTT = (t4 - t1) - (t3 - t2)$.

Some synchronization protocols (*e.g.*, SNTP) assume a symmetric forward (*i.e.*, client-to-server) and reverse (*i.e.*, server-to-client) delay. That is, they calculate One Way Delay (OWD) to be half of the RTT. Hence at true time *t2*, the client's time (T_{client}) and offset with respect to *t2* are expressed as $T_{client}(t2) = t1 + owd$ and $offset(t2) = t2 - (t1 + owd)$.

The key takeaway is that the accuracy of the two-way exchange in calculating the clock offset at the client depends on the validity of the assumption that the forward and reverse delays are symmetric and specifically that the forward delay is half of RTT. When this assumption is invalid due to path asymmetry, the delay gets added to the calculated offset as error. To illustrate the issue, consider an example in which the RTT of the exchange is 600 ms, with the forward delay being 400 ms and the reverse delay being 200 ms. If the client's clock is behind the reference by 20 ms, the correct offset that needs to be calculated from this exchange should be +20 ms. Because of the assumption of symmetric forward and reverse delays, however, the forward OWD is calculated as 300 ms with an underestimation error of 100 ms (which is half of the path asymmetry of 200 ms). This error in the calculation of the OWD is reflected in the offset calculation: +120 ms instead of the expected value of +20 ms. Similarly, if the forward and reverse delays are 200 ms and 400 ms, the calculated offset would be -80 ms due to the overestimation of the forward OWD by 100 ms. It can be shown that the error in offset calculation is bounded by 0.5 * RTT.

In wireline hosts, OWD asymmetry can occur due to path dynamism and variable switching delays, among other reasons. Such variability is often more pronounced in wireless hosts due to wireless effects such as interference and channel noise [21]. Given that the asymmetry error is bound by half of RTT, synchronization protocols typically address the error by preferring samples with smaller RTTs over others. Moreover, synchronization protocols treat this as a *statistical variability problem*, hence their filtering approaches require multiple samples to pick the best RTT. For instance, NTP addresses this problem by collecting measurements from multiple reference servers and picking the server with the least RTT dispersion. From all the samples from that server, NTP selects the sample with the smallest RTT. Similarly, RADClock [34], which has been shown to outperform NTP in terms of accuracy, maintains a moving window of measurements and uses a weighted sum of measurements within the window to calculate the offset. Specifically, the samples with RTTs closer to the minimum RTT are weighted more than those that are much larger. Both protocols require multiple measurements for every offset calculation, extensive tuning, and in the case of the RADclock, a lot of state. Approaches like Kalman filtering attempt to address these errors by modeling the clock in order to calculate the offset from noisy measurement samples. In addition to the difficulty of accurately modeling clock hardware, particularly for inexpensive hardware like those used by IoT devices, it is also computationally intensive, limiting scalability. In addition, Kalman filtering has been shown to exhibit degraded performance in the presence of non-Gaussian outliers [19].

Assumptions

Our architecture assumes that: *(1)* IoT devices are capable of connecting to a cloud server, and *(2)* IoT protocols support heterogeneous protocol stacks. Our architecture accommodates both WiFi-enabled devices that can connect directly to the cloud or that can function as a bridge.

To support a wide range of IoT devices, we do not assume availability of unlimited compute power and/or processing capability. Although our experiments target specific IoT platforms, we assume that the IoT device may have a very low-quality oscillator (*e.g.*, ceramic instead of crystal [31]) and a limited energy source. Lastly, we assume that IoT devices can

respond to incoming packets and can adjust their clocks based on offset values sent to them. In our ongoing work, we are examining how to efficiently and accurately synchronize devices that are intermittently connected, or may be disconnected over long durations. Finally, we do not specifically address security issues in this paper. Similar to other protocols, we assume that IoT devices will initiate time synchronization to a specified set of servers, which provides a minimal level of assurance. We posit that additional modes of packet exchanges, including secure hashes (*e.g.*, to detect man-in-the-middle attack) and encryption could be used without affecting the accuracy or correctness of our synchronization approach. We intend to develop a more complete security model in future work.

Applicability of SPoT for IoT deployments

Given the design goals and the architecture of SPoT, we argue that it is applicable in a wide range of IoT applications and we describe several deployment scenarios that would be able to utilize our architecture. Consider a smart cloud manufacturing (S-CM) [30] shop floor environment that requires tight clock synchronization between sensors, controllers and tools in order to assure quality and productivity and provide real-time monitoring. Further, to gain insights into telemetry data fused from globally distributed manufacturing sites, time synchronization with a global time reference such as coordinated universal time (UTC) becomes crucial; SPoT is an ideal candidate in such a scenario. Another example is earthquake detection IoT networks [2, 10] that use cloud-connected devices to detect earthquakes in real-time. Tight time synchronization with a global time source enables the development of sophisticated real-time earthquake detection algorithms, and such low-cost IoT networks could utilize SPoT for accurate time synchronization. Yet another example is IoT deployments that utilize public blockchains to ensure trust and accountability [14, 12, 13]. Blockchain implementations require time synchronization for protocol correctness [7, 11] and preliminary code inspections reveal that popular blockchain implementations use SNTP [1] which is shown to perform poorly under noisy network conditions (see below and [21]). In such deployments, SPoT can facilitate accurate time synchronization.

TIME SYNCHRONIZATION IN IoT ECOSYSTEM

Experimental setup

IoT devices. In our experiments we use four different Arduino MKR1000 devices (to compare instances of the same device), a commonly used platform for IoT prototyping [3], and one Raspberry Pi3 (to contrast with the Arduino). In all our wide area experiments, we use Amazon's AWS IoT cloud, and the message broker is colocated with the cloud server.

Testbed. Our testbed includes three components: *(1)* a wireless access point (WAP), *(2)* IoT device, and *(3)* a monitor node (MN, Macbook pro laptop). The IoT device connects to the IoT cloud through the WAP. The MN is connected to the IoT device over a serial port interface and is used to collect timestamps (and other relevant statistics) from the device. In the experiments described below, the NTP-corrected system clock of the MN is used as the reference clock to benchmark the internal system clock of the IoT device. That is, the MN is synchronized with `0.pool.ntp.org` before the start and throughout the duration of experiment, which is 1 hr. We run the experiment for 24 hrs to collect clock offset (for all devices), which we use in our trace-driven analysis for testing SPoT's scalability (§SPoT Evaluation).

Drift characteristics of clock hardware

To gain perspective on IoT clock synchronization we examine the drift characteristics of different Arduino hardware instances. We also consider clock drift under different ambient temperature conditions. Our experiments gather a pair of timestamps every second. The first timestamp is obtained using the NTP-disciplined system clock of the monitor node. The second is a *raw* millisecond counter value obtained from the IoT hardware's clock over the serial port. We then calculate clock offsets using the timestamps measured from the IoT device's raw counter and from the NTP-corrected monitor node. We run the experiments at three different locations: *(a) L1*: a temperature-controlled server room maintained at $\sim$14 °C, *(b) L2*: an office room setting at $\sim$21 °C, and *(c) L3*: a residential apartment at $\sim$27.5 °C. We consider these temperatures as representative of common real-world IoT deployments.

Figure 1 shows the calculated clock offsets throughout the experiment for different hardware instances at locations L2 (left) and clock offsets of the same hardware instance at all three locations L1, L2 and L3 (middle). Visually, we observe that different hardware instances of the same prototyping platform show quite different drift characteristics and drift rates and hence different values of clock offset for the same duration of the experiment. Further, even the same hardware instance exhibits different clock drift rates depending on the ambient temperature. The plots show that the difference in behavior due to the differences in ambient temperature is apparent even for small durations (*i.e.*, 10 min). The variability in drift characteristics implies that *accurate synchronization will be challenging for any mechanism that expects a uniform drift characteristic from all IoT devices*. Moreover, for mechanisms that expect to model the clock hardware and that involve the use of training data, we posit that clock synchronization will be resource intensive due to the need to model/train on a wide range of drift characteristics at different temperatures [18, 17].

Stability of clock hardware

Next, we compare the stability of clock hardware on the Arduino devices with the Raspberry Pi3. A typical measure of oscillator stability (and hence clock stability) is the *Allan variance* [27]. Given a series of consecutive offset measurements of a clock at certain measurement interval τ, the fractional frequency or instantaneous clock skew is the rate of change of offset calculated using consecutive offset measurements. The Allan variance is an estimator of the variance of the clock skew for the given measurement interval. Since the rate stability of a clock is a function of τ, often the square root of the Allan variance called the *Allan deviation* is plotted as a function of τ on a log-log curve to study clock stability [34].

The Allan deviation curve of typical PC clock hardware when viewed on a log-log plot is often made up of two lines: *(a)* the line with a slope of -1, dominated by white phase noise, which is the measurement noise characteristic of the channel; and *(b)* the line with slope of +0.5, dominated by the random walk noise, which is characteristic of the clock wander. The intersection of these two lines—which is also the inflection

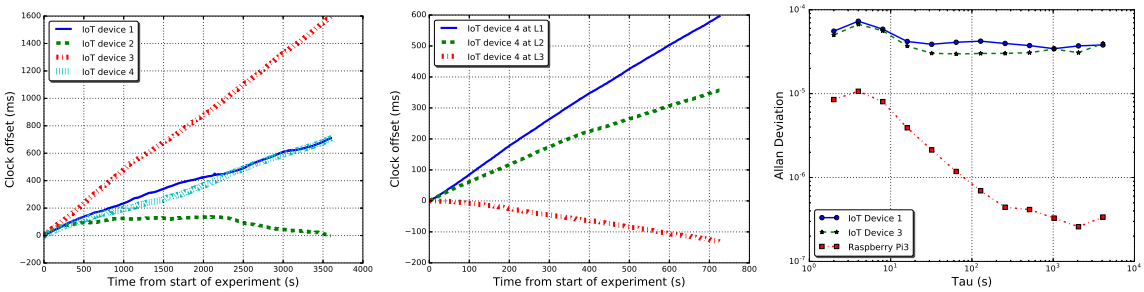

Figure 1: **Comparison of drift characteristics of different Arduino devices (left) and same Arduino device under different ambient temperatures (middle) and comparison of Allan deviation plots of Arduino devices 1 and 3 and Pi3 (right).**

point in the curve—is called the *Allan intercept*. The Allan intercept characterizes the given clock hardware and network path: that is, it represents the measurement interval where the error due to the measurement noise from the network path and the frequency error due to the wander of the clock would be minimal. Thus, it is an important statistic that influences the design of rate synchronization and polling behavior of many time synchronization protocols such as NTP [28] and RADClock [34]. These protocols expect to see the intercept in the range of $\tau = 1000$ s.

Figure 1 (right) shows the Allan deviation plots for the Raspberry Pi3 and two Arduino devices (*i.e.*, IoT devices 1 and 3) used in our experiments, all using offset measurements collected at location L2. The shape of the Allan deviation curve and the Allan intercept around 1000 s for Pi3 is consistent with prior studies[24]. From the figure, it is clear that the stability of Arduino clock hardware is much lower and does not satisfy the stability assumptions made by other clock synchronization protocols. The Allan deviation is in between 10^{-4} and 10^{-5}, that is, the variability of the clock frequency/rate is in the range of 10s of ms even for short measurement intervals, which is consistent with our earlier observations (see Figure 1). Also the flatter Allan deviation curves indicate that the clock wander effects start to dominate much earlier that 1000 s. It should be noted that the Allan deviation of Pi3 is slightly higher than the one seen in [24], which is simply because of our measurement setup where we measure Pi3 offsets from across a LAN using the monitor node in our setup discussed in §Experimental setup—an observation consistent with prior studies [28]. From Figure 1 (right) it is evident that IoT hardware do not meet the design assumptions used by synchronization protocols like NTP, RADClock, etc., signaling the need for synchronization mechanisms that can accommodate lower clock stability and diverse clock drift characteristics. This lower observed stability also implies that the estimation of Allan intercept as part of the synchronization mechanisms would be quite challenging and calls for simpler methods to estimate the stability of the clock and to pick suitable measurement/polling intervals. These insights are key to our proposed approach for rate synchronization and varying the polling interval.

SYNCHRONIZATION PROTOCOL FOR ioT

SPoT Architecture

Our synchronization algorithms are designed to be lightweight and scalable. The basic design components include software that runs on clients and a server infrastructure that runs in a cloud environment. To support IoT devices with different computational capabilities, our architecture allows two types of synchronization clients namely, *thick* clients and *thin* clients. Thick clients run the lightweight synchronization algorithms with the polling interval between synchronization requests, which are determined by the algorithms. The server simply responds to the timestamped message exchanges initiated by these clients. For thin clients with limited compute capabilities, the synchronization algorithms are run on the server. Thin clients respond to timestamped message exchange requests sent from the server and use the offset and the clock skew provided by the server. Further, the energy limitations of the IoT devices can be addressed by choosing different polling regimes and by setting the Error Margin (EM) accordingly. Beyond the core components, our system includes algorithms for offset and rate synchronization, described below.

Offset synchronization in SPoT

The key to achieving good offset synchronization is to directly address asymmetry errors. This is accomplished by identifying the *direction* (*i.e.*, is the asymmetry on the forward or the reverse path?) and *magnitude* of the asymmetry. Once these have been identified, SPoT corrects for the asymmetry error in the offset calculation. To identify the direction of the asymmetry for a given measurement, we consider how the offset is affected by a particular measurement. Given an expected offset, from the earlier discussion, it is clear that an asymmetry in the forward direction (*i.e.*, client-to-server) increases the expected offset by an amount equal to half of the magnitude of asymmetry; similarly, an asymmetry in the reverse direction (*i.e.*, server-to-client) decreases the offset by an amount equal to half the magnitude of the asymmetry. Hence by comparing the computed offset to the expected offset, the direction of the asymmetry can be inferred. This insight forms the basis of SPoT's filtering approach. The magnitude of the asymmetry is estimated to be the difference between the minimum RTT of all samples seen so far and the RTT of the current sample.

SPoT uses the offset synchronization algorithm to correct the error introduced by asymmetric path delays. Given the clock skew, last known offset and its measurement time, the algorithm calculates an estimate of the current clock offset and the magnitude of the asymmetry. If the measured offset is significantly greater than the estimated offset, the asymmetry is determined to be on the forward path and the measured offset is corrected accordingly. Similarly, correction is also applied for reverse path asymmetry. If both the tests are not satisfied, the additional delay is inferred to be a symmetric additional delay and the measured offset is accepted without

any correction. The test to check if the measured offset is significantly greater or lesser than the estimated offset is based on a user tunable threshold called the EM, which is set to 10ms in our experiments.[3] Further details of the offset algorithm can be found in [20].

Rate synchronization in SPoT

An important input in our system is the clock skew, which is required for calculating an estimated clock offset. For devices with lower-quality clock hardware that can drift significantly between synchronization points (see §Time Synchronization in IoT Ecosystem), the clock skew is also necessary for correcting for the clock drift when reading time on the device between synchronization points. Estimating and updating the clock skew is the problem of *rate synchronization*. SPoT's rate synchronization algorithm works in conjunction with its offset estimation algorithm.

The accuracy of the calculated clock skew depends on the stability of the clock hardware and the duration between subsequent measurements. For stable clock hardware, the calculated clock skew will remain valid and accurate for longer durations and hence the polling intervals could be large. For hardware that is less stable, the clock skew should be updated more often and hence the reference should be polled more frequently. Further, the accuracy of the clock skew has an impact on the accuracy of the offset algorithm. SPoT's rate synchronization algorithm uses this insight to determine the stability of the clock hardware and picks the best possible polling interval between measurements.

The rate synchronization is run every time the offset algorithm is executed. The frequency of running the offset algorithm and hence the frequency of polling or measuring the clock offset is controlled by the polling interval as determined by the rate synchronization algorithm, which calculates the absolute error between the estimated offset and the corrected offset for every offset synchronization point. For an observation time (set to 5 minutes in our experiments) or at least until 5 such absolute errors have been observed, which ever is longer, the algorithm calculates a running mean of these absolute errors. Once the observation time has expired, the mean absolute error is compared against the EM. If the mean absolute error is less than twice the EM, the clock is determined to be stable and hence the polling interval is increased. Similarly, if the mean absolute error is greater than twice the EM, the clock is deemed to be unstable and the polling interval is decreased accordingly. Further, higher quality samples, where the measured offset and corrected offset are same, are used to update the clock skew. The amount of increase or decrease applied to the polling interval depends on the polling style chosen by the user.

The polling style could be Additive Increase and Multiplicative Decrease (AIMD) or Multiplicative Increase and Multiplicative Decrease (MIMD) depending on the accuracy requirements and energy budget of the device. For devices with lower energy budget (*e.g.*, operating on battery power) and lower accuracy requirements, MIMD could be used. MIMD is aggressive in increasing the polling interval in order to reduce the number of offset synchronization measurements for slightly reduced accuracy. EM could also be increased by the user to further decrease the polling cost incurred by the synchronization algorithms in lieu of synchronization accuracy. Further details of the rate synch algorithm can be found in [20].

[3]This threshold was determined experimentally and proved to be robust across our evaluations. We do not include a sensitivity analysis due to space limits.

SPoT Implementation

To accommodate both thick and thin clients, SPoT implementation consists of four major components: *(1)* the core library implementing synchronization algorithms for thick clients, *(2)* the scalable reference server implementation that supports both thick clients and thin clients, *(3)* a reference implementation for a thin client, and *(4)* a client emulator that can support multiple thin clients on a single physical node, used in our scalability experiment (see §SPoT Evaluation). The core library is implemented in ∼400 lines of C and can be directly used by thick clients and the reference server implementation. The library is designed to be lightweight even for thick clients and maintains only 15 variables to manage the synchronization. SPoT's lightweight implementation makes it well-suited for resource-constrained IoT platforms, some of which do not allow tasks to run in a dedicated thread [9]. Existing synchronization systems that run in dedicated daemon processes [5, 34] would simply not be possible on such platforms.

SPoT EVALUATION

Experimental Approach

To test the synchronization accuracy of SPoT and to compare SPoT to other protocols such as SNTP, MQTT, and filtering methods used by MNTP [21] and `ntpdate` [8], we add *observational noise* to the collected offsets and perform trace driven analysis. We compare SPoT with these protocols due to their widespread adoption in the Internet [21] and the preference among developers to use these protocols for a surprisingly wide range of scenarios from popular mobile applications [4, 6] to blockchain clients [1]. We use a trace driven approach since the addition of noise in live tests is extremely difficult to control. The observational noise is drawn from a normal distribution with zero mean and a given standard deviation, which we set at different levels in our experiments. To ensure that both noisy and noiseless offset observations are equally probable, we add the observational noise to each offset measurement in the trace with probability 0.5. To generate RTT measurements corresponding to the noisy offset measurements, we begin with the observation that the two-way method halves the actual noise and the actual noise is simply additive delay on top of the path RTT. That is, we start with a given path RTT and add twice the absolute value of the corresponding observational noise as an additive delay. The path RTT which is set to 300 ms[4] in our analysis can be set to any nominal value without affecting the results of the analysis. We do not consider network errors such as packet drops, duplicates etc., since the mechanisms to handle these errors such as retries and timeouts are implementation details that do not affect the correctness or accuracy of our algorithms. We do not consider scenarios like extended periods of loss of connectivity and plan to address this as part of our future work.

Using the traces, we compare SPoT to other protocols at three levels of noise: low, medium, and high; where the standard

[4]We selected this value to approximate the RTT between an IoT device and a cloud-based reference server.

deviations of the noise distribution are set to 50 ms, 150 ms, and 250 ms respectively. We run the different synchronization protocols on the 24-hr trace and collect statistics including RMSE, minimum, maximum, and standard deviation of offset errors. In our results, we report error statistics averaged over 100 runs. In all of our experiments, we run SPoT with AIMD polling behavior and with error margin set to 10 ms. Moreover, in addition to comparing SPoT with widely-used IoT-specific synchronization protocols, we also compare SPoT with two other mechanisms. (1) Consensus is the offset filtering method used in [21] for bootstrapping the synchronization mechanism with high quality measurements. To report each offset, the Consensus method makes 8 measurements, each 15s apart. Outliers are eliminated and the average of the remaining measurements are used. (2) MinRTT is the filtering approach used by the standard `ntpdate` implementation where an offset measurement with a minimum RTT among 8 samples is selected [8]. In our evaluations we do not compare thick and thin implementations since they offer the same accuracy.

SPoT vs. other protocols

Offset errors. Table 1 compares the RMSE among different synchronization protocols at different noise levels. The values in parenthesis are the RMSE values obtained from raw offset measurements without any filtering. Since SNTP and MQTT have no filtering capabilities, their offsets are the same as raw offsets. From the table we observe that SPoT performs consistently well by maintaining the same level of accuracy ($\sim$10 ms), even under high noise levels, where as the accuracy of other protocols deteriorate with increasing noise levels. Specifically, SPoT performs 16x, 19x, and 22x better than MQTT in low, medium, and high noise levels. Similarly, SPoT provides 3x (low), 10x (medium) and 17x (high) better accuracy versus SNTP. Of all the protocols examined, MQTT has the lowest clock synchronization accuracy followed by SNTP. Although Consensus and MinRTT methods perform well with low noise levels, their accuracy is affected by increases in noise levels.

Table 1: **Comparison of synchronization RMSE (ms) values at different noise levels.**

Protocol	Low noise	Medium noise	High noise
SPoT	10.0 (35.6)	9.3 (102.1)	8.9 (173.4)
SNTP	36.2 (36.2)	105.5 (105.5)	161.7 (161.7)
MQTT	162.5 (162.5)	196.6 (196.6)	225.7 (225.7)
Consensus	9.1 (34.5)	27.0 (109.8)	42.5 (175.7)
MinRTT	8.7 (34.8)	21.8 (107.0)	41.9 (177.5)

Figure 2 shows the offsets reported by SPoT for the first hour of the experiment under high noise level for IoT devices 1 (top) and 2 (bottom). From this figure, we first observe that SPoT is effective in correcting offset measurements in the face of high noise levels, whereas the unfiltered offsets produced by SNTP are as high as 600 ms. Second, we see that clock skew estimates produced by SPoT's rate synchronization algorithm follow the original ground truth offset, despite the different non-linear clock drift trends of devices.

We also compared the performance of SPoT, MNTP, and SNTP using a separate trace-based experiment. In this experiment, we assume the client is running in an IoT bridge environment since MNTP makes certain assumptions about clock drift that make it inappropriate to use directly on an IoT device that may exhibit non-linear drift [21]. The trace used was of SNTP packet exchanges every 5s, used in our prior work [21]. Since SPoT uses a variable polling interval, we interpolated the raw SNTP measurements by using the same value for any time during a given 5s interval. The maximum synchronization RMSE (ms) for SPoT, MNTP, and SNTP, was 0.72, 6.172, and 51.89, respectively, showing that SPoT's algorithms provide a significant boost in accuracy over the other techniques.

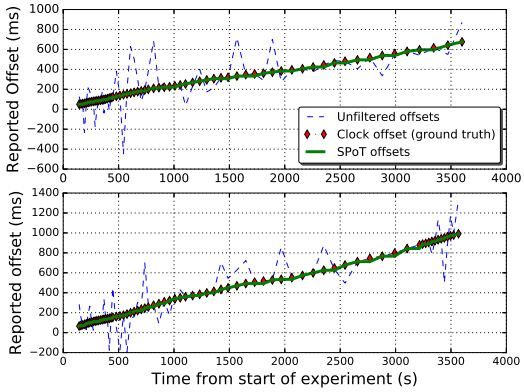

Figure 2: **Offsets of IoT devices 1 (top) and 2 (bottom) reported by SPoT from the first hour of the experiment with high noise level.**

Error statistics. To complement Figure 2, Table 2 shows the minimum, maximum and standard deviation of offset errors for the different synchronization protocols for high noise level. We make two key observations: *(1)* since MQTT is a push-based mechanism that simply publishes the timestamps to IoT clients, a major source of error is the uncorrected OWD in the published timestamps; and *(2)* none of the synchronization mechanisms, except SPoT, provide rate synchronization and simultaneous variation of polling interval based on stability of device's clock (*i.e.*, they run with a default polling value of 128 s). The minimum error of 0 ms for all protocols except MQTT is due to their ability to identify/use noiseless measurements in/from a noisy environment. While Consensus and MinRTT protocols reduce the maximum error with respect to the raw unfiltered offset measurements, it is clear that SPoT is more effective: it bounds the maximum error to be within 50 ms. Furthermore, the standard deviation of offset errors for SPoT is lower than other protocols (*i.e.*, within 10 ms).

Next, we calculate the rate errors at each synchronization point by estimating an offset using the clock skew provided by SPoT in comparison with the ground truth offset value. Hence these errors provide a bound for worst case offset errors incurred by using the clock skew estimates that are produced by the rate synchronization process. We observe that SPoT's rate synchronization is able to achieve RMSE values of 14.7 ms, 13.3 ms and 13.5 ms under conditions of low, medium and high noise. It is clear that SPoT's rate synchronization accuracy is consistent under all noise levels (*i.e.*, rate errors are less than 15 ms). Since the rate synchronization mechanisms vary the polling interval depending on the stability of the clock hardware, we note that SPoT's polling is adaptive and robust to all noise levels.

SPoT's polling behavior. Figure 3 compares the AIMD and MIMD polling behaviors of SPoT on the Arduino hardware. The figure shows that MIMD is more aggressive in increasing

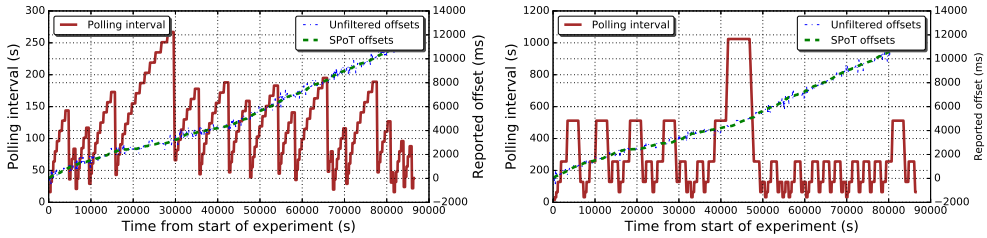

Figure 3: **Comparison of polling behaviors of SPoT on Arduino hardware with AIMD (left) and MIMD (right).**

the polling interval. The RMSE error incurred by AIMD is 8.9 ms while the RMSE for MIMD is 14.7 ms. The number of offset measurements made by AIMD for a period of 24 hr is 953 and 545 for MIMD. This shows that IoT devices that have a strict energy budget but lower synchronization accuracy requirements should opt to use MIMD instead of AIMD.

Table 2: **Comparison of offset error statistics under high noise level.**

Protocol	Minimum (ms)	Maximum (ms)	Standard Deviation (ms)
SPoT	0.0 (0.0)	47.5 (771.36)	7.8 (142.4)
SNTP	0.0 (0.0)	709.0 (709.0)	133.8 (133.8)
MQTT	150.0 (150.0)	781.7 (781.7)	107.6 (107.6)
Consensus	0.0 (0.0)	253.1 (896.6)	32.3 (144.9)
MinRTT	0.0 (0.0)	417.7 (855.1)	40.2 (147.6)

Similarly Figure 4 shows the difference in behavior between AIMD and MIMD for SPoT running on the Raspberry Pi3. The relatively stable clock hardware of the Pi3 can be seen from the total clock drift of about 200 ms for a period of 24 hr compared to a drift of about 12,000 ms on the Arduino hardware. As designed, SPoT is able to exploit the relatively higher stability of Pi3 hardware and reduce both offset and rate sync RMSE even in the presence of high noise. The RMSE for offset synchronization is 1.5 ms for AIMD and 3.0 ms for MIMD. Similarly, the rate synchronization RMSE values are 2.9 ms and 3.0 ms for AIMD and MIMD respectively. It is also clear that given the relative stability of Pi3, SPoT is effective in increasing the polling interval; MIMD is able to reach the maximum polling value of 1024 s very rapidly so the number of offset measurements made by AIMD for a period of 24 hr is 269 and that of MIMD is only 132.

Table 3: **Memory and CPU profile of SpoT server.**

No. of clients	1	10	100	1 k	10 k	100 k	1 M
Instruction count	669 k	699 k	719 k	1 M	4 M	34 M	332 M
Execution time (ms)	0.02	0.02	0.04	0.18	0.72	6.53	64.07
Memory (B)	7 KB	7 KB	7 KB	33 KB	312 KB	3 MB	30 MB

Scalability of SPoT. We conduct a series of tests in the wide area to examine the scalability of SPoT. We note that our implementation uses a single node to serve all thin clients, which we use as a baseline. We use two cloud nodes, one on the east coast and another on the west coast of the U.S. One of them runs the SPoT server, while the other runs the client emulator. Since the SPoT server maintains no state for thick clients its operation is essentially the same as the NTP reference server, thus we examine server scalability for thin clients only. Both the server and the client emulator have their clock disciplined by NTP. That is, their expected offset throughout the experiment is 0 ms. We use several benchmarking runs, each with 1, 10, 100, 1k, 5k, 10k and 15k clients. In each run, we synchronize thin clients with the server for 5 min and calculate the average RMSE of offsets.

In our detailed results (not shown due to space constraints), all clients had avg. RMSE < 2 ms, indicating that synchronization accuracy remains consistent as the number of clients grow. We expect similarly consistent results with larger number of clients as they are deployed across multiple servers.

To understand the resource consumption of SPoT we conduct micro-benchmark experiments that measure the CPU and the memory usage. Table 3 shows the average number of machine instructions and execution time (ms) required by the server to complete one round of synchronization for different numbers of clients when running on a Ubuntu 17.10 server with a quad-core 1.8 GHz Intel i5-3337U processor and 3.7 GiB memory, averaged over 100 runs. Total memory required by the SPoT server for different number of thin clients is also shown, which includes both the state information used by SPoT's synchronization algorithms as well as the bookkeeping information required by the server to keep track of all clients. From the table we observe that the SPoT server's footprint is light on the CPU (execution time and instruction count) and memory usage, even for a high number of clients.

Table 4: **SPoT server throughput (PPS) required to support thin client.**

No. of clients	1	10	100	1 k	10 k	100 k	1 M
Arduino-AIMD	0.01	0.1	1	11	110	1.1 k	11 k
Pi3-AIMD	0.003	0.03	0.3	3	31	311	3.1 k
Arduino-MIMD	0.006	0.06	0.63	6	63	630	6.3 k
Pi3-MIMD	0.001	0.01	0.1	1.5	15	152	1.5 k

Finally, using the number of packets exchanged by different polling methods of SPoT for Arduino devices and Pi3 for a period of 24 hr, we estimate the server throughput (packets/sec (PPS)) required to synchronize different number of thin clients. From Table 4 we can see that the network overhead to run the SPoT server is low, with only a required throughput of about 6k PPS for 1M Arduino devices and about 1.5k PPS for stable hardware such as Pi3.

RELATED WORK

Our work relates most closely to prior studies that have examined environmental effects on oscillator performance and clock drift, as well as synchronization protocols for sensor network platforms and other constrained environments. Schmid *et al.* [32] extensively evaluate the problem of clock drift in low-end oscillators in a variety of conditions. In a somewhat similar vein, there are a number of unpublished investigations by IoT prototypers and hobbyists (*e.g.*, [31]) who provide anecdotal evidence of the effects of environmental conditions on different types of oscillators available on Arduino-based IoT platforms. These studies inform our work by highlighting the challenges of addressing drift on low-end devices.

Wireless sensor networks (WSNs) typically consist of a large set of relatively homogenous nodes, with significant energy,

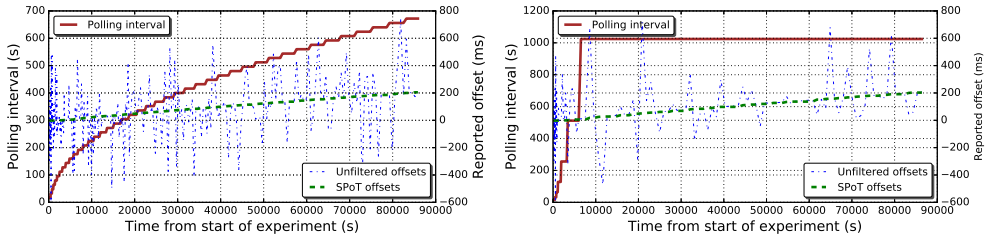

Figure 4: **Comparison of polling behaviors of SPoT on Pi3 hardware with AIMD (left) and MIMD (right).**

bandwidth, and computational constraints. While IoT devices are typically constrained in different ways than WSNs, there have been a number of time synchronization methods developed in the WSN context (Sundraraman *et al.* provide a survey of these methods in [33]) that have a bearing on our work, including [15, 16, 22]. The protocols developed for WSN domain exploit characteristics of the broadcast MAC layer to avoid network inconsistencies that cause time synchronization errors. Hence, to synchronize to a global timescale such as UTC, these techniques require a UTC time source to be part of the same broadcast domain. Finally, Kalman filters have been used in the context of time synchronization in order to model clock offset and skew, and to handle missing information [18, 17]. These methods that model clock hardware become highly challenging and resource intensive at scales introduced by the IoT domain, due to huge variability in drift characteristics exhibited by IoT hardware under different ambient temperature conditions, as discussed in §Drift characteristics of clock hardware and §Stability of clock hardware.

SUMMARY

In this paper, we consider the question of how to synchronize clocks in an Internet of Things. We begin by investigating clock drift in two standard prototyping platforms over a range of operating conditions that would be typical for an IoT device. We find clock drift on the order of seconds over relatively short time periods. This level of variation makes standard protocols such as SNTP and those based on MQTT ineffective. We address this problem by developing a new architecture for synchronizing clocks on IoT devices. We develop a prototype implementation of our design to evaluate efficacy over a range of configurations, operating conditions and target platforms. Our results show that SPoT outperforms MQTT and SNTP by a factor of 22 and 17 respectively, in the presence of high noise levels, and maintains a clock accuracy of within 15ms at various noise levels. Finally, we report on the scalability of our server implementation through microbenchmark experiments and show that our system can scale to support large numbers of clients with minimal resource utilization.

Acknowledgements

This work is supported by NSF grants CNS-1703592, DHS BAA 11-01, AFRL FA8750-12-2-0328. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, DHS, AFRL or the U.S. Government.

REFERENCES

1. Fast, light, robust Ethereum implementation. Link. (????).
2. IoT: Sensing Earthquakes before hand with Grillo. Link. (????).
3. Maker Madness: The Best IoT Boards of 2016. Link. (????).
4. 2010. SNTP implementation for iOS. Link. (2010).
5. 2014. NTP Daemon. Link. (2014).
6. 2016. Snapchat coding error nearly destroys all of time for the internet. Link. (2016).
7. 2017. How do I set the time to be synchronized on Parity? Link. (2017).
8. 2017. ntpdate Documentation. Link. (2017).
9. 2017a. Particle Docs - system threads. Link. (2017).
10. 2017. ShakeAlert: Implementing Public Earthquake Early Warning for the U.S. Link. (2017).
11. 2017b. Your clock is not in sync. Link. (2017).
12. 2018a. Blockchain IoT - IBM Watson IoT. Link. (2018).
13. 2018. Hyperledger - Open source blockchain for businesses - IBM Blockchain. Link. (2018).
14. 2018b. Using blockchain to secure the internet of things. Link. (2018).
15. J. Elson and others. 2002. Fine-grained Network Time Synchronization Using Reference Broadcasts. In *Usenix OSDI*.
16. S. Ganeriwal and others. 2003. Timing-sync Protocol for Sensor Networks. In *ACM SenSys*.
17. B.R. Hamilton and others. 2008. ACES: Adaptive Clock Estimation and Synchronization using Kalman Filtering. In *ACM Mobicom*.
18. H. Kim and others. 2012. Tracking Low-precision Clocks with Time-varying Drifts using Kalman Filtering. *IEEE/ACM TON* (2012).
19. J. Levine. 2016. *IEEE Trans Ultrason Ferroelectr Freq Control* 63 (Jan-04-2016 2016), 561 – 570. DOI:Link
20. S.K. Mani and others. 2018. A System for Clock Synchronization in an Internet of Things. Link. (2018).
21. S.K. Mani, R. Durairajan, P. Barford, and J. Sommers. 2016. MNTP: Enhancing Time Synchronization for Mobile Devices. In *ACM IMC*.
22. M Maróti, B Kusy, G Simon, and Á Lédeczi. 2004. The Flooding Time Synchronization Protocol. In *ACM SenSys*.
23. K. Marzullo and S. Owicki. 1983. Maintaining the Time in a Distributed System. In *ACM PODC*.
24. P. Membrey and others. 2016. Time to Measure the Pi. In *ACM IMC*. DOI:Link
25. D.L. Mills. 1981. DCNET Internet Clock Service. Link. (April 1981).
26. D.L. Mills. 1985. Algorithms for Synchronizing Network Clocks. Link. (1985).
27. D.L. Mills. 1996. *The network computer as precision timekeeper*. Technical Report. DELAWARE UNIV NEWARK DEPT OF ELECTRICAL ENGINEERING.
28. D.L. Mills. 1998. Adaptive hybrid clock discipline algorithm for the network time protocol. *IEEE/ACM Trans. Netw.* 6, 5 (Oct 1998), 505–514.
29. S.B. Moon, P. Skelly, and D. Towsley. 1999. Estimation and removal of clock skew from network delay measurements. In *Proceedings of INFOCOM'99*.
30. Lei S.P. Wang Z.Z. Qu, T. and others. 2016. *Int J Adv Manuf Technol* 84, 1 (01 Apr 2016), 147–164. DOI:Link
31. J. Rantwijk. Arduino clock frequency accuracy. Link. (????).
32. T. Schmid and others. 2008. Exploiting Manufacturing Variations for Compensating Environment-induced Clock Drift in Time Synchronization. *ACM SIGMETRICS* (2008).
33. B. Sundararaman and others. 2005. Clock Synchronization for Wireless Sensor Networks: A Survey. *Adhoc networks* (2005).
34. D. Veitch and others. 2004. Robust Synchronization of Software Clocks Across the Internet. In *ACM IMC*. DOI:Link