Low-latency Synchronization of Loosely-coupled Sensornet Republishing

USC/ISI Technical Report ISI-TR-660b [*]
Original: Mar 2009 / Revised: May 2010

Unkyu Park and John Heidemann
{ukpark,johnh}@isi.edu
Information Sciences Institute
University of Southern California

Abstract

Today many individual deployments of sensornets are successful, but they will have much greater impact when, rather than standing alone, they share data across deployments so each can build upon the others. We expect data to be shared over the Internet, and as the number of processing and reprocessing steps grows, *timely data synchronization* is increasingly important. Today, such sharing is often hard-coded or driven by fixed-interval polling. Fixed-interval polling can provide poor worst-case performance (mean latency approaching the data publishing period), and best performance requires careful manual configuration of both poll period and phase. It is more difficult to find polling schedule when multiple sources of different rates are used. We instead propose *Data Publication Tracking* (DPT), a new family of adaptive polling algorithms that *learn and predict* good times to pull data to minimize both latency and unfruitful queries. DPT coordinates how consumers of data contact sources, identifying rendezvous times that are most likely to yield new data from a single or groups of sources. Our approach avoids manual configuration and automatically adapts to outages and changes in data publishing rate of multiple sources. To evaluate our work, we examine four sensornet deployments and develop a rough *model of sensornet data publishing*. We then use this model and replay of real traces to evaluate DPT, finding that, depending on application, its median latency is only 10–30% of that of fixed-interval polling, with a configurable rate of network load that is the same or slightly higher.

1 Introduction

Sensor network applications have been proposed and are helping scientists with their research [25, 24, 27, 12]. As a new, automated instrument, sensornets enable collection of data that has previously been too expensive to acquire in areas of micro-climate monitoring [25], animal habitat [24], geology [27], and similar areas [12]. These deployments are undertaken by different research groups, each to accomplish their own specific objective. While these research groups often make their data available, reuse of data is rare, and collaboration across multiple sensornets is even rarer. Even ignoring issues of data ownership, today it is too cumbersome to easily share data, even for scientists studying overlapping subject areas.

Towards sharing: We anticipate that *data sharing* represents the next phase of sensornet development. Our goal is not just to allow single projects to interconnect isolated sensor network patches, but to allow different research groups to easily share their data. Moreover, broad participation and interest often arises when the barrier to sharing sensor data is sufficiently low that casual users and amateurs can participate and share data, fostering the *citizen scientist* [21]. (In fact, amateurs already share data in some science [26] and commercial applications [14].)

In the limit, we expect individual users will share sensor data, the blogging-inspired vision of *slogging* or sensor logging first described by Mark Hansen [2]. The result will be a *Sensor-Internet* that blends sensornets and the Internet; a topic just now being explored by several groups [21, 15, 18].

While individual sensors are sometimes of interest, the data becomes much more compelling when it is aggregated and processed—we call such processing *republishing*. Much as important results "bubble up" from distributed cross-linking in blogs, we seek a similar milieu of sensor data. We expect data from many different sources comes together to form a rich world where sensor values are checked against each other, filtered, corrected, combined and divided, and indexed, not just by the sensor owners but potentially by anyone with access to the data. We have described our first steps towards this approach previously [21, 20].

Figure 1 shows an example that we currently run, with the loose, multi-step processing we hope will become more common. Temperature data is generated from dozens of sources, from mote-like sensors, PCs, or "scraped" from websites, and each stream is published to a sensor log or *slog*. Two different users correct different kinds of errors (Republishers 1 and 2). Finally, a third republisher fuses data from many different streams to produce an evolving temperature map for the region (Republisher 3). Here, sensor data propagate through four users.

Challenges and our contributions: Distributed, stream

[*] This paper extends the ISI Technical Report 660, first released in March 2009. This revision adds new cover details of Bimodal DPT (Section 6.3 and 7.6) and Multi-source DPT algorithm (Section 6.5 and 7.7 and Algorithm 3). The text in some sections slightly has been expanded to provide more detail.

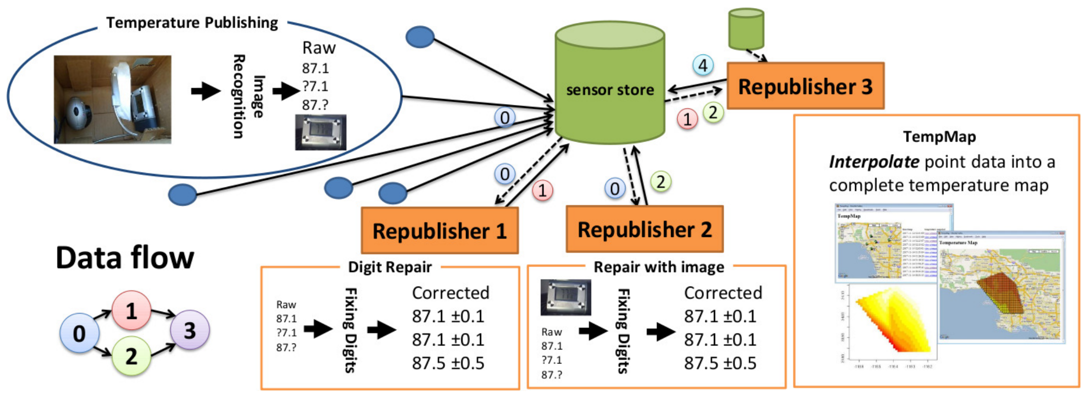


Figure 1. A sensor-sharing example: data is published, republished with corrections, then republished again into a map.

data immediately raises the question of *synchronization* between sites: how do different sites assure getting the correct data rapidly? Fortunately data is usually immutable and timestamped, so consistency is not a problem. The primary problem is *data latency*: how do different sites know when fresh data is available? how do users get new data as quickly as possible? And as we show in Section 4.1, current approaches add delay each processing step, making latency intolerable if the processing pipeline exceeds a few steps. When data is time-sensitive, such as reports on traffic or weather conditions, user satisfaction relates directly to processing delay. Without our approaches, traffic congestion reports could not use more than a few processing steps. And these questions must be considered not just for simple data pipelines (single source, single result), but also for consumers that fuse data from multiple sources.

Our paper proposes Data Publication Tracking (DPT) to support low-latency synchronization for distributed processing of sensor data (Section 6). We place all tracking costs with the data consumer, because in a large distributed environment it is not possible for publishers to track all consumer. Today, ad hoc systems often simply pull for data at fixed intervals, causing cumulative latency if consumption is not carefully tuned (Sec. 4.2). Instead, we show that each consumer can model data publishing and predict when new data is likely to arrive. These predictions provide much lower latency than fixed-interval polling, and they reduce the fraction of unfruitful queries (Sec. 7). We also provide Multi-source DPT that coordinates multiple sources and finds the time when the rendezvous of some or all of the sources become available.

Although tracking the data source is conceptually simple, we find that the details of the data model have a significant effect on the savings we obtain; in Section 6.1 we show several small variations that result in significant performance differences (Section 7.2). A second challenge is that long-term data collection systems accommodate changes in data publishing over time and outages. We evaluate how Disruption-Tolerant Networking approaches interact with synchronization (Section 7.6). Finally, while with effective synchronization, consumers ask for data immediately after it is generated, but in very large systems this synchronization creates a "thundering herd" of requests that can swamp a publisher. We propose modest, intentional de-synchronization to mitigate this problem (Section 6.4).

We wish to evaluate our approaches against typical sensor networks traffic. However, to our knowledge there are today no published models of long-term sensornet deployments. We therefore develop an approximate model from more than a year of four different long-running sensornet deployments. Although nominally each system publishes data at regular intervals, we find the realities of outages and clock drift require a more sophisticated model (Sec. 5.1). We validate our model against these four deployments, finding that it is far from perfect, but much closer to reality than a simple periodic model (Sec. 5.2).

Finally, we use this traffic model and replays of the four deployments to quantify the benefits of improved synchronization in Section 7. While fixed-interval polling can do reasonably well, particularly when manually tuned to the data source, it can have very bad worst-case performance (mean latency equal to the polling period) if tuned to match the period but out-of-phase with the source. The specific results depend on the application, its median latency is only 10–30% of that of fixed-interval polling. Variants of DPT can be selected to prefer reducing latency or network overhead. DPT-L, optimized for low network overhead, gets around 10-50% the latency of fixed and generates 5-35% fewer unnecessary server requests. A moderate configuration, DPT-N, gets 5-10% of the latency but with about 5-17% additional unnecessary requests.

The main contributions of this paper are to explore how data publication tracking can provide low-latency distribution of sensor data across network of republishers, and to develop the first traffic models from real, long-term sensornet deployments.

2 Related Work

Our work builds on previous efforts to share sensornet data over the Internet, and is inspired by RSS-style sharing in Internet blogs. It is also similar to workflow in large scientific applications. We examine each of these areas next.

2.1 Sharing Sensor Data over the Internet

Several research efforts are exploring how sensornets and the Internet can interact: IrisNet [9], SenseWeb [19, 29, 15], GSN [1], Simple Sensor Syndication [3], Sensor-Internet [21].

IrisNet considers Internet-side storage of XML-tagged data from PC-connected sensors [9]. It uses a hierarchy of XML databases to enable search over fields. They recognize the need for distributed administration of storage but do not provide a solution. SenseWeb is a software infrastructure for sharing multiple sensor streams and exploration of environmental measurement. It allows users to publish heterogeneous sensor data and share them with others. Our work has similar goals as IrisNet, particularly in distributed management, and we share SenseWeb's goals of sharing sensor data on the Internet, and we could build on SenseWeb's storage and visualization capabilities. Both IrisNet and SenseWeb, however, pull data at fixed intervals. We instead develop an adaptive polling algorithm to reduce data latency. We also emphasize the need for multiple levels of data processing with republishing.

GSN is a middleware design to integrate heterogeneous sensor networks [1]. GSN provides an abstraction of sensor network that separates the sensor data from the specific hardware and software used in the sensor network. GSN stores sensor data at *GSN nodes* and provides data processing specified by SQL. A peer-to-peer network indexes sensor data, allowing efficient discovery of GSN nodes based on data type and range. Both our work and GSN assume there will be multiple processing steps; we propose adaptive polling will reduce latency relative to GSN's fixed-interval polling.

Simple Sensor Syndication places sensor data over RSS and has shown how users can act in response to these feeds [3]. Although sensor data can be accessed by any RSS reader, they do not discuss data timeliness. Again, we believe RSS would benefit from our adaptive polling approach; our approaches should be applied to their RSS-based mechanisms.

Finally, in Sensor-Internet [21] we show the components of a system for sharing and searching Internet data. We later extended this system to track data use as it is processed and republished [20]. However we have not examined latency reduction until now.

We think synchronous data retrieving can benefit all Sensor-Internet systems by detecting and delivering new sensor data rapidly.

2.2 RSS Feeds Retrieval and Aggregation Problems

Sharing real time sensor data over the Internet has much in common with sharing blog entries.

Sia *et al.* exploit the change characteristics of RSS postings in fine-grained time and apply a stream specific polling policy to detect new RSS feeds rapidly [23]. They showed that an adaptive polling schedule monitors RSS feeds efficiently. We also pull the source data with a polling policy customized to each RSS stream to reduce the detection latency of new data. While they find the optimal timing of pull for a given number of pulls per period, our polling policy tracks the each data point and predicts the time that the next one will be published. The main difference in our tracking approaches is that blogs show wide variance in publish times because content is usually human generated. Sensor data, however, is often fairly regular as we show in Section 5.

Most RSS readers poll the server regularly. A common practice is to poll at the top of every hour, concentrating traffic and harming performance [5]. Several approaches have been proposed to improve scalability of RSS servers to many clients. First, they try to reduce the unnecessary polls with a larger update interval. Most RSS readers today use a fixed update interval, independent of the stream update rate. The server may also provide a Time-To-Live (TTL) to clients, an estimate of when new data is likely to arrive [28]. Other researchers have suggested replacing HTTP-based RSS feeds with other approaches, such as distributed hash trees. While these approaches reduce overall load, none directly surges due to synchronized traffic. In Section 6.4 we propose use of intentional jitter in client request rates for sensor streams; this same approach would work well for RSS streams.

Stream Feeds [6] brings the sensor data to Internet using a simple abstraction that is expressed in URL. They allow users to access both historical data and the real-time updates with selective push-and-pull retrieval method. They use a push-based delivery mechanism for new data. Push-based mechanisms provide very low latency, but we believe they place an unacceptable burden on the data creator. We therefore explore synchronized, pull-based methods.

Yahoo Pipes is a web-based GUI tool for creating custom processing of RSS feeds, including aggregation and filtering [30]. It is therefore closest to our concept of multiple steps of republishing. Unlike our work, though, they may assume a centrally managed compute infrastructure, while we believe a distributed scheme is essential if sensor sharing is to spread beyond a single company.

2.3 Scientific Workflow

Scientific workflow allows scientists to access and analysis of massive scientific data which are typically distributed, and heterogeneous [22, 16]. Workflow systems today support large scientific computations, often on large grid computers. Most workflow systems assume analysis of large, stored, and static datasets. With such datasets, synchronization is not critical since there are few items to synchronize and they are ready at computation start. We instead focus on streams of sensor data that evolve continuously and depend on good synchronization for low latency.

3 Background about Sensor Sharing over the Internet

As described above in related work (Section 2.1), there are several current proposals to share sensor data over the Internet. Our approaches could build upon many of these existing systems. Since each of the existing systems use different terminology for similar concepts, we define the terminology we use in this paper. As we introduce our terminology we discuss our assumptions about the sensor data sharing ecosystem. We then introduce four sensornet deployments that provide datasets and motivate our work.

3.1 Components of Sensor Data Sharing

Figure 2 shows several scenarios for how users might share sensor data over the Internet.

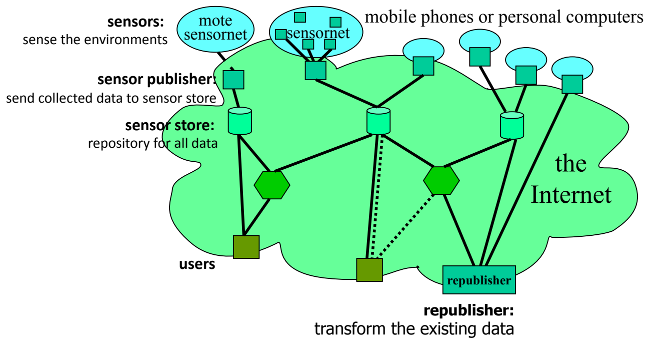


Figure 2. Components of sensor data sharing over the Internet.

Our basic assumption is that *sensors* generate data, while *publishers* connect sensors to the Internet. If sensors run a custom, non-IP network protocol, publishers function as gateways. If not, the sensor and publisher may be logical functions that are combined on a single machine. Figure 2 suggests that sensors could be traditional mote networks, full IP-connected sensors (possibly also motes [11]), individual Internet-connected PCs, or even mobile telephones.

On the Internet, data is kept in sensor data stores, or *sensor stores* for short. Our central assumption is that there will be *multiple* sensor stores. We believe diversity is essential if sensing is commonplace; and anticipate both large, centralized stores that host data for many users, and small stores run by individual researcher projects or hobbyists. Such diversity is common in both web and blog hosting, and we already see it emerging with some sensor data.

Finally, in addition to publishers who post sensor data directly, we anticipate *republishers* that process existing data and make the results available to others. The essential element of republishers is that they make their analysis available by placing it in a sensor store just like direct publishers. With this step we hope to enable value-added analysis of sensor data with examples such as data aggregation, filtering, statistical estimation, vetting, and error suppression.

3.2 Sensornet Deployments

While the behavior of sensor publishing seems trivial, we find that real deployments provide surprising behavior. In this paper we consider datasets from four different deployments as shown in Table 1.

Our primary dataset is *Temperature*, an urban temperature monitoring deployment running for 12 months [21]. Data comes from two classes of sensors: most are connected to personal computers operated by individuals, others are data collected indirectly from websites that report data from professional or hobbyist weather stations. One artifact of the diversity of sensor owner is that sensors provide very different reliabilities: some running for months like clockwork, and others attached to laptops providing data only for a few hours at a time. Data is often processed through multiple steps to clean up errors as shown in Figure 1. This dataset best matches our model of a successful sensor sharing ecosystem, since data comes from many independent providers and there are several republishing steps.

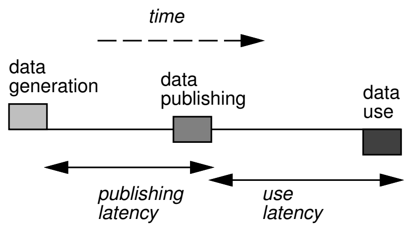


Figure 3. Events and latencies in publishing.

Our second dataset is *Habitat*, a multi-year deployment monitoring the habitat of a nature reserve [24]. Sensors measure environmental conditions (temperature, humidity, etc.) and cameras collect images of animals in their homes. Unlike Temperature, this dataset is centrally managed and curated.

Third, we consider *Seismic*. This data represents data from seismic sensors watching for earthquake activity [13]. PC-class sensors are deployed over a long (more than 300km) transect, forwarding data over point-to-point 802.11 and ultimately through a DSL gateway to the Internet. From there, data passes through two processing steps at different institutions. This deployment uses an implementation of Disruption Tolerant Networking, both to handle forwarding outages and to batch transmissions over the DSL line. We selected this dataset because of its use of DTN and its 4-step processing.

Finally, we consider *Location*, a small (2-sensor) deployment. PC-class sensors track their location with GPS and relay data to the Internet using a simple DTN-like system built with rsync. We selected this dataset to observe an alternative implementation of DTN.

4 Republishing Challenges

Researchers have made significant progress in the mechanisms for sharing sensor data over the Internet (see Section 2.1). However, we see three significant challenges to enabling an ecosystem of sharing: republishing efficiently, managing long-term faults, and characterizing sensor traffic.

Efficient data sharing should minimize network usage and propagate data quickly. Figure 3 shows the three events help characterize data latency: data is *generated* at the server, it is *published* to an Internet-connected data store, and then it is *consumed* by a user or a republisher. To minimize overall latency, we must separately consider *publishing latency*, the time from data generation to publishing on the Internet, and *use latency*, the time from availability via publishing to consumption by a user.

We next show the motivations of republishing synchronization and see how republishing synchronization can reduce use latency, how disruption-tolerant networking affects publishing latency, and how we need models of sensor traffic to evaluate both.

4.1 Republishing Synchronization Motivations

The essence of republishing is *many independent groups* creating and processing data. This diversity of users requires a *loosely coupled* system of republishing. In integrated sys-

Dataset	Description	Disruption Tolerant	Sensors	Duration (months)
Temperature	temperature monitoring in an urban area [21].	No	5–20	12
Habitat	wildlife habitat monitoring [24]	Yes	16	12
Seismic	seismic monitoring over a 300km transect [13]	Yes	45	30
Location	long-duration GPS monitoring	Yes	2	2

Table 1. Sensornet deployment datasets considered in this paper.

tems run by a single organization, a data publisher can often coordinate with data consumers, either pushing or pulling data. We instead minimize the effort imposed on the publisher, with the goal of minimizing publisher costs and so encouraging publishing.

In a sensornet republishing system, rapid use of new data makes decisions based on the current status of the physical world. Timeliness is particularly important in sensor-actuator systems, where sensed data triggers additional actions, or when a detection prompts human evaluation of the data and potential modifications to the system. For example, off-line data about vehicle traffic on a roadway supports long-term planning, but low-latency data can help drivers choose routes to avoid delays today. Even in disciplines like ecology, where datasets span months or years, rapid feedback about data is essential to ensure that data collection is valid and ongoing.

Our goal of reducing latency is particularly important when there are *multiple steps*, each doing republishing. With fixed-interval polling, each step can easily accumulate delay. As shown in Fig. 1, we expect the sensor data are reprocessed by others and the result are also used by another republisher. With fixed-interval polling, each step accumulates half of the polling period as delay, so in this four step example, temperatures would typically be 20 minutes old if data was taken every 5 minutes.

Sometimes republishing fuses results from *multiple sources*, each with potentially different periods and phases. It is hard to find the optimal schedule of accessing each source and running the republishing process.

Although very frequent polling is an easy means to reduce latency, it wastes network traffic and CPU resources on both the sensor store and republisher. While unnecessary polling is of little concern with powerful servers, efficient polling is essential to reduce the cost of sensor sharing by allowing one sensor to support many users and republishers.

4.2 Republishing Synchronization

How should a consumer coordinate with its publisher of sensor data? Direct triggers are possible in centrally controlled systems, but in loosely coupled systems a consumer often must poll the publisher.

In many cases sensor data is generated regularly, perhaps every minute or hour. It is natural to assume a data consumer would therefore also query for data at a fixed rate. Figure 4 shows the mean latency for fixed-period polling of a source generating new data every 300s. (This data is generated artificially using the model we describe in Section 5.1). First, we show how the fixed interval polling policy works. The penalty of this simple approach is that each step incurs, on average, latency equal to half of the polling period. In

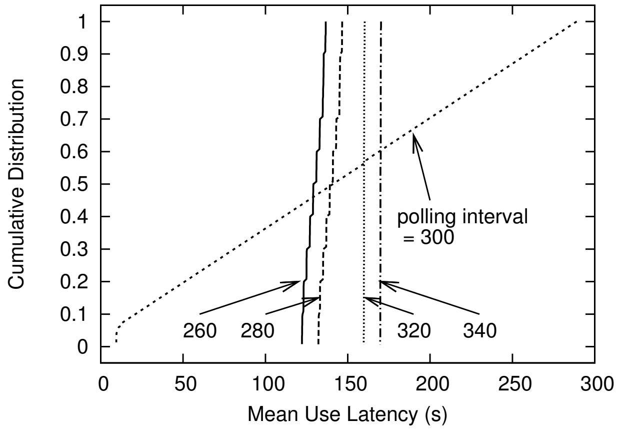


Figure 4. Mean latency for fixed-interval polling of data generated every 300s, as polling interval and phase vary (p_{ss} = 0.95, p_{fs} = 0.80)**.**

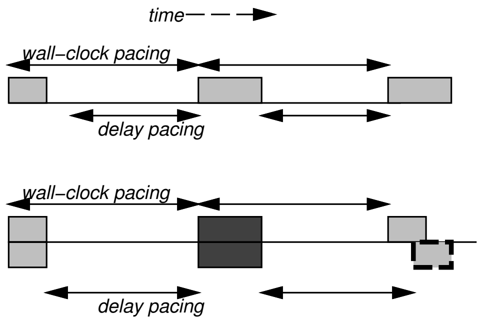


Figure 5. Two implementations of periodic data collection, wall-clock and delay pacing, with fixed collection times (top) and varying collection times (bottom).

a multi-step republishing world, this *poor synchronization* causes significant processing latency.

One could improve latency by polling for updates very frequently. Yet frequent checks waste network and CPU resources—while polling 300s data every 5s provides only 2.5s mean latency, it means that 98% of all requests are unnecessary!

A better idea would be to poll at the same period in which data is generated. Polling data every 300s, just after it is generated, gives near-zero latency. However, it is essential to match not just the period, but also the *phase* of data publishing—polling just before data is generated gives the worst-case latency of 299s for every observation.

While the effort of synchronizing period and phase seems sufficient, real deployments are much more difficult. Periods may change if the system is reconfigured; such changes must be updated at all consumers. Phase may have to be re-synchronized if either publisher or consumer reboot. If data publishing is temporarily delayed due to high load, the correct phase for the consumer may change. Figure 5 shows how subtle implementation differences can cause variation. Pacing can be implemented by relating sampling to fixed intervals of wall-clock time (*wall-clock pacing*), or by fixed delay between samples (*delay pacing*). With delay pacing, the exact timing depends on the time it takes to collect observations and the system hardware. In the bottom example of Figure 5, a longer collection time for the middle sample (the dark boxes) causes the periodicity of delay pacing to vary, and to diverge from wall clock pacing (compare the timing of the dark dashes sample at the bottom right to samples with wall-clock pacing or consistent sample times above). For all these reasons we conclude that system management can be significant for fixed-interval polling.

Because of these challenges of high latency, unnecessary network usage, and significant management cost, we believe fixed-interval polling is difficult for a single sensor and consumer, and untenable with many sensors and multiple levels of republishers. We therefore propose *Data Publication Tracking* (DPT) to provide *adaptive synchronization*. We explore DPT in Section 6.1.

Finally, with either manual or adaptive synchronization, a successful system will have many data consumers. If all consumers are perfectly synchronized with the publisher, load at the publisher becomes very bursty—consumers become regular flash crowds. We therefore also propose *intentional de-synchronization* to spread load when necessary in Section 6.4.

4.3 Disruption Tolerant Networking

The goal of many sensornet deployments is delivering data to scientists, so sensornets are often designed to cope with network and server outages. Delay/Disruption-Tolerant Networking is an approach to data transfer where intermediate nodes buffer and retransmit data to mitigate large delays and disconnection [7]. Here we consider DTN as an approach that encompasses many implementations, ranging from the Disruption Tolerant Shell [17] to manually copying data with physically carried disks ("sneakernet") [10].

DTN affects publishing latency, the time between data generation and when it appears in an Internet-based data store. Figure 6 shows distributions of publishing latency for the three datasets we consider that use DTN-approaches for publishing. All deployments show a long tail where some data arrives well after it is generated; this data would be lost in a system without DTN. Yet the deployments see very different mean publishing latencies: less than a minute, two hours, a day, and even a month for the sneakernet subset of Seismic.

DTN poses a challenge to efficient data sharing because data appears in bursts at unpredictable times. In addition, sensors shift between on-line, regular updates to off-line, batched updates. These challenges motivate our bimodal model of publisher tracking in Section 6.3.

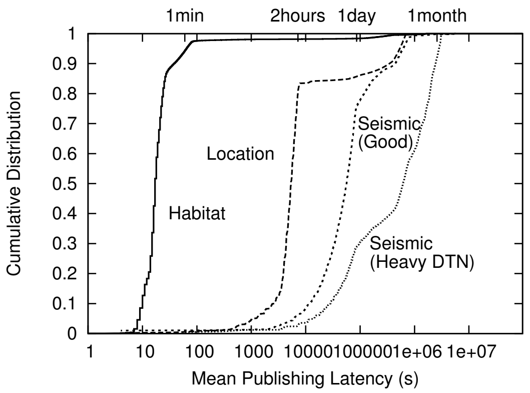


Figure 6. Mean publishing latency (time from data generation to Internet availability) for three datasets.

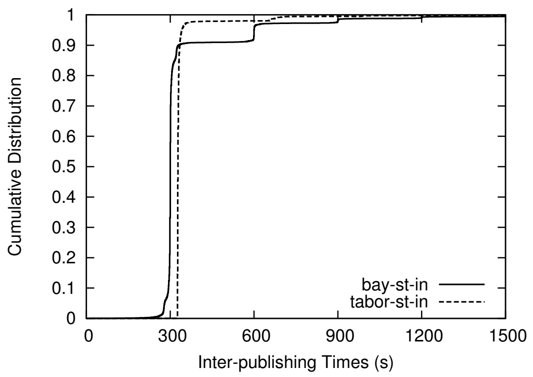


Figure 7. Distribution of inter-publish times (time between publish events) for two sensors (dataset: Temperature).

4.4 Need to Understand Sensor Data Streams

To improve synchronization and to cope with artifacts due to disruption-tolerant networking, we must understand the sensor data stream.

Many sensornets target regular sensing intervals. Ideally, temperature taken every 300s will appear every 300s. However, in real deployments, many things conspire against such regularity; delays occur due to system or network load, outages occur due to failures, and observations drift due to imperfect clocks and software changes and restarts. The Temperature dataset targets 300s collection intervals, yet Figure 7 shows it is far from perfect—we see some jitter around the nominal 300s interval, and some percentage of reports are missing, resulting in interarrivals at multiples of 300s. We see similar variation in our other deployments.

We therefore next develop *models of sensor traffic* to understand what real-world traffic looks like. We use these models to drive our synchronization algorithms and generate artificial traffic.

5 Modeling Sensor Publishing

Our goal is to track and predict the flow of a sensor data stream from a publisher. We next develop a model of sensor publishing, then show that it provides a reasonable fit to the data streams in the four deployments we consider. We will use this model in Section 6 to develop a predictive synchronization algorithm.

Our basic model is inspired by Figure 7: we expect some jitter in each interval, and some lost reports. We formalize this model below, and use it in our synchronization algorithm.

We have also observed that long-term outages are a third component of sensornet deployments. Long-term outages have many root causes. In the four deployments we see outages from hours to months, from reasons including laptop-connected sensors, software troubles, and failure of inaccessible sensors. Because of this diversity of causes we do *not* try to model long-term outages.

We do not intend our model to be perfectly accurate, but merely good enough to support better synchronization. After we present the model, we quantify its accuracy in Section 5.2. We also later extend it to support bimodal distributions in Section 6.3.

5.1 Formalizing the Publishing Model

Our publishing model captures two aspects of publishing: jitter around expected arrival times, and short-term failure to report. Jitter is usually due to load at the sensor, network, or data store; it can occur randomly or systematically as described in Section 4.2. Short-term outages are often due to sensor malfunction, network outages, brief sensor or datastore maintenance or reboots.

One can express inter-publish times as:

$$I = (k+1)\lambda + j \tag{1}$$

where k is the number of consecutive failures, λ is the target publishing period, and j represents random jitter.

The simplest possible model is to assume there are no failures. That is, $k = 0$. We call this model the *non-failure model.*

If we assume a fixed probability of success to publish, p_s, then k, the number of consecutive failures is a random variable with a geometric distribution:

$$P(k; p_s) = (1 - p_s)^k p_s \tag{2}$$

We call this model the *geometric model.*

Although we began with this simple geometric model, we found it provided a poor match to real-world deployments because failure is *not* completely independent. Even ignoring long-term failures, we see runs of lost sensor values. One cause would be maintenance on a sensor or data store that lasts longer than the sensing period. Instead, we see that when data is published successfully, the next data is also likely to be published, while if the previous data was lost, the next is less likely to be successful.

We model this correlation with a two-state Markov chain, where the states are the success or failure of the last publish attempt:

$$\mathbf{P} = \begin{pmatrix} p_{ss} & (1-p_{ss}) \\ p_{fs} & (1-p_{fs}) \end{pmatrix} \tag{3}$$

where p_{ss} is probability of success after a successful publish, and p_{fs} is probability of success after failure.

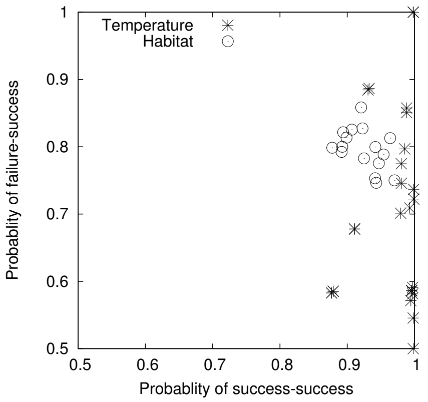


Figure 8. Correlation of data publishing probability for each sensor in two deployments (Temperature and Habitat).

Figure 8 shows the correlated loss probabilities for all sensors in two deployments. We see that p_{ss} is consistency higher than p_{fs}, showing that it often takes some time to recover after failure. We also see that most Habitat sensors have similar loss characteristics, while Temperature sensors have a wider range of reliability. This variation corresponds to centralized and independent sensor ownership and operation.

Our complete sensor model therefore incorporates correlated loss:

$$P(k; p_{ss}, p_{fs}) = \begin{cases} p_{ss} & \text{for} \quad k = 0 \\ (1-p_{ss})(1-p_{fs})^{k-1} p_{fs} & \text{for} \quad k > 0 \end{cases}$$

We use this *two-state Markov model* for simulations of sensor data and to motivate our synchronization algorithm. To get artificial traffic that models a given deployment we compute p_{ss} and p_{fs} from long-term traces.

As described above, we do not attempt to model long-term failure. We define long-term failures as consecutive failures of more than m publishing attempts, currently setting $m = 5$. To prevent long-term outages from polluting our estimates of short-term failures, we ignore outages longer than m when computing empirical values of p_{ss} and p_{fs} from a given sensor deployment.

Now we consider the publishing jitter, j. Prior work has shown Laplace distributions provide a good model of network jitter [8, 31, 32, 4]. Although publishing jitter includes processing components as well as jitter due to the network, we find a Laplace distribution fits our observations well. The PDF of jitter model is therefore:

$$j \sim \frac{1}{2b} e^{-\frac{|x-a|}{b}} \tag{4}$$

where a is the mean of the jitter, $b = \sqrt{\sigma^2/2}$, and σ^2 is the variance.

5.2 Model Accuracy

In this section we evaluate our models against alternatives using our deployments.

In each case we compute our correlated failure model to fit the data, then generate artificial sensor data of the same duration using our model compare against the real trace. Since the parameters are computed from the data we expect some fit, but our model is much similar than reality. We compare two components of the model: the number of consecutive failures, and jitter around the target publish rate.

First, we evaluate the accuracy of the failure component of the model. We assume an interarrival time which is much longer than expected indicate a missed publish attempt, so we can then count the number of consecutive publishing failures by analyzing the interarrival times. Let the i th published data arrive at time T_i, so the interarrival time of i th data is $I_i = T_i - T_{i-1}$. We then compute the number of failures by considering an interarrival in the range from $(h + 0.5) \times \lambda$ to $(h + 1.5) \times \lambda$ to represent h consecutive failures where λ is the expected publishing interval. We compare the number of consecutive failures between model and traces from deployments we observe. We define a correct model fit using the Chi-squared test statistic with 0.05 significance:

$$Error_{failuremodel} = \sum_{h=0}^{m} \frac{(Model_h - Observed_h)^2}{Model_h} \qquad (5)$$

where m is the maximum number of consecutive failures that we considered. $Model_h$ and $Observed_h$ are the number of h-consecutive failures in the model and observed data.

We compare our proposed two-state Markov failure model with our two simpler models, non-failures and and the geometric (uncorrelated) failure model. The failure error of simple model is not mathematically stable because the $Model_h = 0$ for $h > 0$, so we add a single failure to prevent a division by zero.

Figure 9 shows the CDF of failure errors of each sensor in the Temperature and Habitat datasets. We first observe that the two-state model is a statistically good fit only for about half of the temperature dataset, and for none of the Habitat sensors (comparing the Chi-squared threshold against the error for sensors in each model). This result suggests that outage durations are not geometric (as per our two-state model), but follow some other distribution.

However, this analysis strongly suggests that our two-state model is significantly better than either of the simpler models. Although inaccurate, our model is close enough to serve as inspiration for our synchronization algorithm.

Second, we evaluate the jitter component of our model. To isolate jitter, we measure how much each publication differs from a fixed period (λ, both empirically and with either a normal or Laplace distribution.) We fit each model using maximum-likelihood estimation of parameters.

We show data for jitter around λ and 2λ for a single sensor in Figure 10. (Other sensors in that and other datasets are generally similar.) We can see visually that the Laplace distribution is a better fit than Gaussian. The Kolmogorov-Smirnov (K-S) distance between empirical data and the normal distribution is 0.1985, while with Laplace it is 0.1475. Although neither distribution is a statistically strong match (they reject the null hypothesis), Laplace is the closer.

We found similar results examining jitter around the other failure multiples (from 2λ to 5λ). Figure 11 shows the jitter

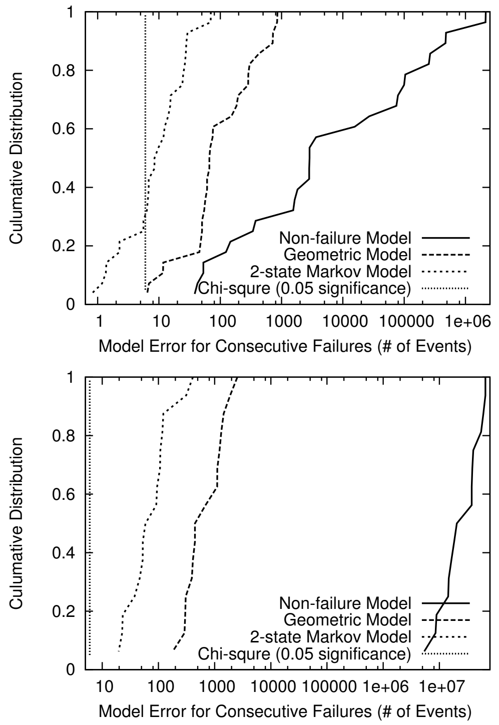


Figure 9. Evaluation of accuracy in modeling consecutive failures from Temperature (top) and Habitat (bottom).

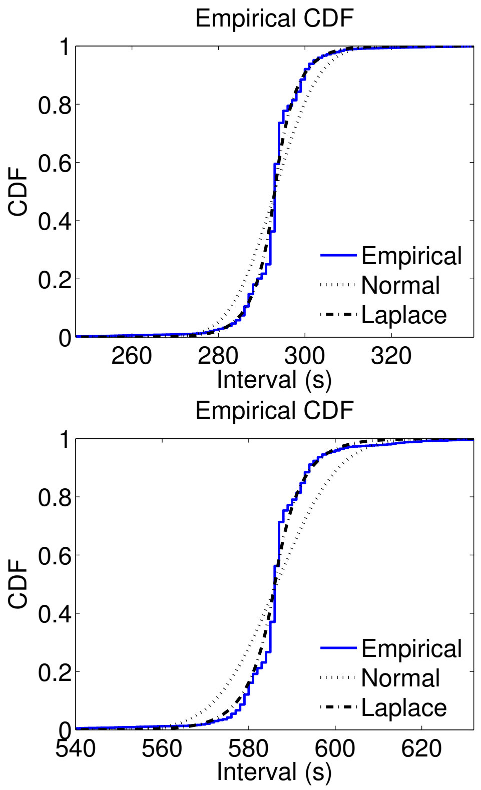


Figure 10. Jitter model fit to the real data (dataset: Habitat, sensor: 9NN): jitter around λ (top) and wider jitter around 2λ (bottom).

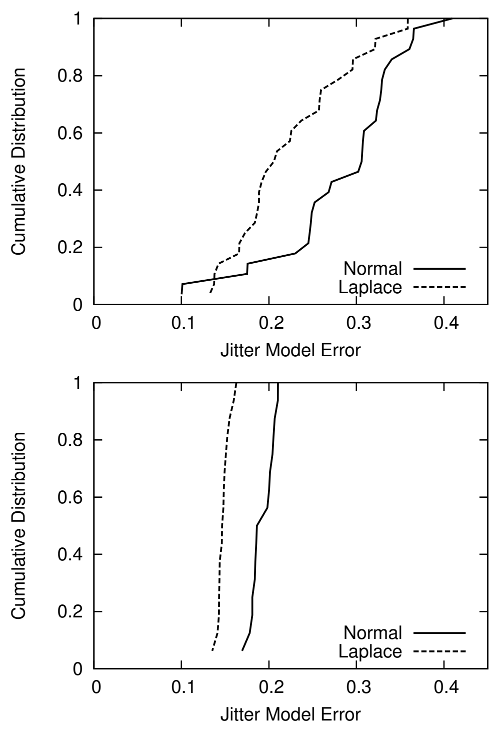


Figure 11. Jitter model error in Temperature (top) and Habitat (bottom).

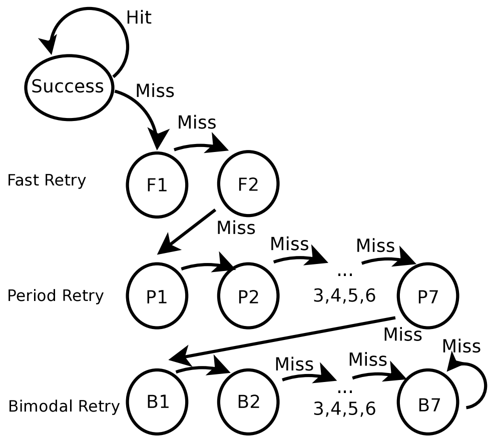


Figure 12. State diagram of DPT-A (Hit always go to *success* state).

model error of every sensor stream in both Temperature and Habitat datasets. In almost all cases, Laplace fits better than normal. Overall, we conclude that our model captures jitter accurately.

6 Synchronization Algorithms

We next describe *Data Publication Tracking* (DPT), our algorithm to improve synchronization. We first present the basic algorithm with several small extensions. We then present Bimodal DPT to support sensornets distributing data with disruption-tolerant networking. Finally, we present Intentional Desynchronization, a load distribution strategy for very popular publishers.

6.1 Basic Data Publication Tracking (DPT)

Since we cannot know for certain when future data will appear, the basic idea of DPT is to track prior publication history to make a best effort prediction of when new data is likely to arrive. Our goal is to improve on the latency of simple period polling, while not producing too many poll *misses*—requests that find no new data present.

Our overall approach is inspired by our observations from sensor data streams as shown in Figure 7. Those observations influenced our model of data streams in Section 5 that considers short-term outages and jitter, and that recognizes but does not attempt to capture long-term outages.

In DPT, each client models the median and standard deviation of inter-publish times in the sensor stream. At the top level, DPT predicts that the next sensor value will be published with the same period, simply adding the median inter-publish estimate to the last publish time. We account for jitter in two ways. First, DPT optionally alters its prediction by adding one standard deviation to this estimate, giving data a little extra time to arrive. Second, if DPT finds new data has not yet been published, it does zero, one, or two *fast retries* to see if data arrives after a short delay. If we do not find data after accounting for jitter, DPT assumes we have lost a sample and polls again after the next period, a *period retry*. To account for extended outages, DPT backs off the interval of period retries exponentially. Figure 12 shows how DPT attempt these retries. We describe more detail below.

DPT is composed of channel modeling (Algorithms 1) and adaptive retry (Algorithm 2). However, it also includes several sub-algorithms: back-off, more-data immediate pull, and phase adjustment. In addition, in the next section we explore bimodal DPT to handle deployments that use DTN, and then intentional desynchronization to handle very large numbers of consumers.

The *core DPT tracking and adaptive retry* algorithms are in blocks (1) and (5), respectively. We track the stream by recording publish time median and standard deviation. To compute median, we keep a window of the last 20 publishing times (not shown in the pseudo-code). (We previously used mean inter-publish time instead of median, since times have a long-tail, median better represents the process.) It is important that we track data publish-to-publish times, not use-to-use times (as defined in Figure 3), since use-to-use time will be zero if we catch up on a burst of previously unread sensor values.

Adaptive retry is the long block (5), where DPT counts failures in each mode (fast retry, period retry, and bimodal retry discussed below). Each stage has a different retry time, and a different number of unsuccessful attempts that cause it to fall into the next mode. Fast retries quickly, spaced by observed standard deviation. Fast retries are useful for catching overly aggressive polling in DPT-A and DPT-N, since those variants expect to have query miss rates of 93% or 50% in their goal of minimizing latency.

Period and bimodal retries (but not fast retries) back-off exponentially (steps (6) and (7)). We use exponential back-off to tolerate extended outages, balancing latency after an uncertain down-time with query hit rate. To prevent back-off from becoming excessive, we cap back-off to a reason-

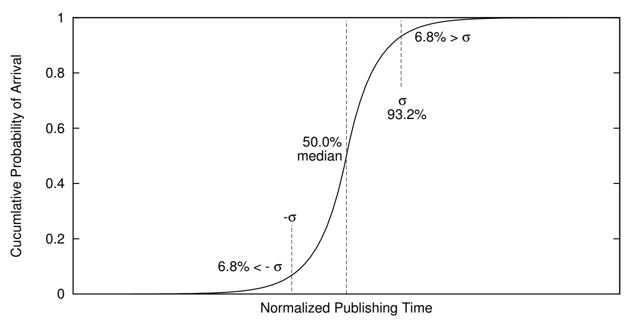


Figure 13. Selection of target polling times in DPT variants against an idealized Laplace distribution.

able value, currently two days (step (8)). Finally, as we move between different back-off modes, we adjust timing with `next_delay_bias` to account for retries done at more rapid modes.

We use *more-data immediate pull* to quickly catch up after multiple sensor values were published. When sensor data is returned, we also signal `result.has_more_data_already()`, allowing the client to immediately request additional results. Such immediate processing occurs when a sensor comes back on-line, or a DTN system delivers a burst of data after a disruption.

Finally, we do *phase tracking* as well as tracking the period of data streams. We expect phase adjustment to correct for sensors that are rebooted or otherwise dramatically change when they report data. Step (4) implements phase adjustment, since after a successful query we "rebase" our timing on the publish time of the data by subtracting the phase adjustment from the current time. Phase adjustment is calculated at the server as a time difference between the data publish time and polling request arrival time. Since we track synchronization only to second-level accuracy, we can ignore propagation delay of the polling request.

6.2 DPT Variants: Latency vs. Hit Rate

We have three *variants of DPT*, *aggressive*, *normal*, and *lazy* (DPT-A, DPT-N, and DPT-L). *Aggressive* attempts to reduce latency by polling one standard deviation before the data is expected, *normal* when it is expected, and *lazy* one standard deviation late. These algorithms therefore trade query misses for lower latency, as we show in Section 7.3. The algorithms are all slight variants of the same code, setting `VARIANT_BIAS` and `VARIANT_FAST_TRIES` to control when polling is done after a success. The more aggressive variants do *fast retries* to account for expected query misses, as we discuss below.

The DPT variants poll at different times relative to the estimated of next data. DPT-N polls at the median, and so by definition it will require a retry 50% of the time; it retries after one standard deviation. Because DPT-A is aggressive, it retries twice before giving up and doing a period retry. Assuming jitter follows a Laplace distribution (our closest fit, and a tighter distribution than Gaussian), analysis shows that about 7% of events occur at $\mu - \sigma$, and 7% after $\mu + \sigma$. Figure 13 shows where we expect each variant to pull.

We considered giving each variant an extra retry. Doing so would reduce latency for 6.8% of the time that is delayed by more than one standard deviation past median, but at the cost of lower hit percentage. Further exploration of this option is future work.

Algorithm 1 DPT: Data Publication Tracking Algorithm

```
Variables:
estimated_median, estimated_standard_deviation
retry_mode = {SUCCESS,FAST,PERIOD,BIMODAL}
retry_count
previous_time
next_delay, next_delay_bias
Constants:
VARIANT_BIAS = -1, 0, 1
VARIANT_FAST_TRIES = 2, 1, 0 for DPT-A, DPT-N, DPT-L
PERIOD_TRIES = 7
DELAY_CAP = 2 days

  initialization: all variables are zero, retry_mode = SUCCESS
  loop
      wait until previous_time + next_delay + next_delay_bias;
      next_delay_bias = 0;
      previous_time = current_time();
      result = request_data_from_source();
      if ( result == SUCCESS) then
          // *** (1) TRACKING THE STREAM
          success_time = current_time();
          if (retry_mode == BIMODAL) then
              update bimodal_median;
          else
              update estimated_median and estimated_standard_deviation;
          end if
          retry_mode = SUCCESS;
          next_delay = estimated_median + VARIANT_BIAS * estimated_standard_deviation;
          if (result.has_more_data_already()) then
              // *** (2) MORE DATA IMMEDIATE PULL
              next_delay = 0;
          end if
          // *** (3) INTENTIONAL DESYNCHRONIZATION
          next_delay += uniform_random (0, result. desync_interval);
          // *** (4) PHASE ADJUSTMENT
          previous_time -= result.phase_adjustment;
      else
          call the retry_method();
      end if
  end loop
```

6.3 Bimodal DPT

We found that basic DPT works well for several of the deployments that we looked at. However, when we examined Seismic, we found that it published data with two different periods. To minimize their impact on other users of the network, they batch sensor data collected during the daytime and send it all at night. During nighttime hours the publisher sends data from the sensors immediately. This transmission pattern represents one policy made possible by disruption-tolerant networking.

Such bimodal operation is a poor match for basic DPT. It will learn the nighttime pattern, but then repeated miss during the day.

To better manage bimodal publishers, we extended DPT to support bimodal operation. For bimodal operation, we track `bimodal_median` in Algorithm 1 block (1), and fail-over to the bimodal estimate after repeated misses in block

Algorithm 2 DPT: retry method()

```
// *** (5) ADAPTIVE RETRY after failure
if (retry_mode == SUCCESS) then
    // for a failure after a success, try fast retries
    retry_mode = FAST;
    retry_count = 0;
    next_delay = estimated_standard_deviation;
end if
if (retry_mode == FAST) then
    // continue fast retries until we have too many
    retry_count++;
    if (retry_count > VARIANT_FAST_TRIES) then
        // after too many fast retries, try next period
        retry_mode = PERIOD;
        retry_count = 0;
        next_delay = estimated_median;
        next_delay_bias = current_time() - success_time;
    end if// (no backoff with fast retries)
end if
if (retry_mode == PERIOD) then
    // continue fast retries until we have too many
    retry_count++;
    if (retry_count > PERIOD_TRIES) then
        // after too many period retries, try bimodal
        retry_mode = BIMODAL;
        retry_count = 0;
        next_delay = bimodal_median;
        next_delay_bias = current_time() - success_time;
    else if retry_count > 1 then
        next_delay ∗ = 2; // *** (6) BACKOFF
    end if
end if
if (retry_mode == BIMODAL) then
    if (retry_count > 1) then
        next_delay ∗ = 2; // *** (7) BACKOFF
    end if
end if
```

(7). We will show this approach help to reduce poll misses for Seismic in Section 7.6.

6.4 Intentional Desynchronization

The goal of DPT is to synchronize consumers with the publisher to minimize use latency. While minimizing latency for clients, good synchronization *maximizes* the load on the server by concentrating all requests at the same time. This problem has been observed in RSS readers, where many readers poll for content at the top of every hour [5]. As a result, a *successful* stream of sensor data with many synchronized clients will subject itself to the equivalent of a distributed denial-of-service attack for each published data item.

To address this problem we provide *intentional desynchronization* in step (3) of Algorithm 1. Each publisher returns a recommended *desynchronization interval* along with successful data. Clients then intentionally jitter future requests uniformly over this interval, allowing the sensor store to distribute load as required.

We choose to distribute desynchronization from the sensor data store. In principle, data consumers could estimate the need for desynchronization based on poor response times, but such an estimate could at best correct the problem after the fact. The sensor store has exactly the information about load, and can make an informed judgment about how widely load should be distributed. We expect the sensor store to track its incoming request queue and increase the spread as queue time rises.

6.5 Multi-source Synchronization Algorithm

While republishing data from individual sensors is interesting, the real power of a publishing ecosystem comes when republishing brings together data from *many sources*. Republishing results from a single sensor require the republisher to anticipate when the data will be available; republishing data from multiple sources requires the rendezvous of some or all of these sources as soon as their data becomes available. We next explore this kind of *multi-source synchronization*.

When gathering data from multiple sources, when should one update the result? We define the *republishing condition* as a fixed fraction of data sources. (That fraction could be all of them.) Our goal is to republish as quickly as possible while avoiding unnecessary polling.

Our basic multi-source synchronization algorithm generalizes single-source synchronization by tracking the period and phase of each source. Based on individual tracking, it calculates the best estimate of when the republishing condition is expected to be satisfied and polls the sources that are expected to have new update.

The full algorithm is shown as Algorithm 3. First, it uses DPT-L to track each stream, because each source may have different publishing period and phase. We choose DPT-L over other DPT variants, because it provides balanced performance in both latency and hit percentage as shown in Section 7.3. Second, based on these individual publishing patterns, it polls the expected sources at the time when required number of updated sources are expected. If the number of updates is equal to or more than the requirement, it initiates the republishing. Otherwise, it repeats until it meet the republishing condition.

Alternatively it can poll all sources including unexpected ones. This approach may find sources updated earlier than they are not expected to have an update which allows to proceed the republishing and reduce republishing latency. But the chance of getting update earlier than the expectation is low (less than 93% with DPT-L) as shown in Section 6.2. We can take the benefit of extra polling only when the expected sources fail to get an update. It is not worth to poll the unexpected sources because of low probability and conditional benefit. We rather to poll only expected sources which are at least 93% of probability of hit percentage with DPT-L.

Our approach has a problem; as the number of sources increase, the probability that republishing condition get satisfied decreases. An easy solution is using a *candidate* threshold which is slightly higher than the number of source that the republishing condition requires. When it polls the sources, it uses a *candidate* threshold which allows to poll more sources than the republishing condition. After polling sources, it only compares the result with *republishing* condition, which is less than the *candidate* threshold. Because it polls extra sources, the republishing hit percentage can increases. (Algorithm 3 doesn't include this feature, because we want to show how it works generally. But it is fairly easy to add this feature into the algorithm.)

In Section 7.7 we show that multi-source synchronization helps to reduce the latency of multi-source republishing.

Algorithm 3 Multi-source Synchronization Algorithm (MDPT)

Variables:
S : set of potential input sources
S_r : ready set (has been polled and got new data)
S_c : candidate set to poll (expected to have new data, but not yet polled)
S_n : not ready to poll (expected not to have new data yet)
s_i : source stream
$\hat{t}(s)$: the expected arrival time of next data of source s

Constants:
n : the total number of sources
m : the minimum number of sources that are required to initiate a republishing (trigger threshold).

```
initialization:
$S = \{s_1, s_2, s_3, ..., s_n\}$ // all source set
loop
    $S_n = S$; // consider input from all sources
    $S_r = S_c = \emptyset$; // initialize the candidate and ready set
    repeat
        // PREDICT THE EXPECTED TIME OF $m - |S_r|$ UPDATED SOURCES
        sort $\hat{t}(s)$ for all $s$ in $S_n$
        pick $\hat{t}(s_i)$ which is $(m - |S_r|)$th element in ordered set above
        wait until $\hat{t}(s_i)$
        // SELECT EXPECTED SOURCE AS CURRENT CANDIDATES
        for all $s$ in $S_n$ do
            if $\hat{t}(s) \le now$ then
                $S_c = S_c \cup \{s\}$
                $S_n = S_n \setminus \{s\}$
            end if
        end for
        // VERIFY THE ACTUAL UPDATES IN CURRENT CANDIDATES
        for all $s$ in $S_c$ do
            poll the source $s$ and update $\hat{t}(s)$,
            if $s$ has new data then
                $S_r = S_r \cup \{s\}$;
                $S_c = S_c \setminus \{s\}$;
            end if
        end for
    until $|S_r| \ge$ m
    // REPUBLISHING CONDITION IS SATISFIED
    // RE-POLL SOURCES THAT HAVE FRESHER DATA SINCE WE LEARNED THEY HAD SOME DATA
    for all s in $S_r$ do
        if $\hat{t}(s) \le now$ then
            poll the source $s$
        end if
    end for
    compute the republishing function based on $S_r$ with freshest data
end loop
```

7 DPT Evaluation

In this section we evaluate our synchronization algorithm using both the models we presented in Section 5.1, and trace data from the four datasets in Table 1. Our goal is to compare DPT to today's widely-used fixed-interval polling, also evaluate the differences in the DPT variants, and show how disruption tolerance changes our results.

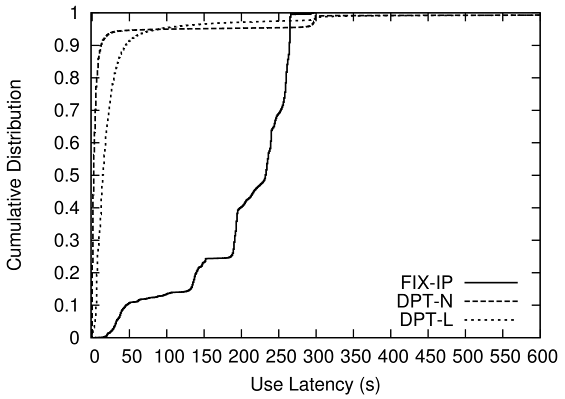


Figure 14. Use latency of all sensor publishings from one reliable sensor (dataset: Temperature, sensor: vir-in).

7.1 Metrics

We evaluate performance with four different metrics, each of which tests slightly different aspects of the performance. In general we evaluate use latency (defined Figure 3); we often drop the "use" when not ambiguous.

Our primary metric is *median use latency*, the 50% percentile value of latency of all data retrievals from a sensor or all sensors in a dataset. We generally prefer median as a statistic over mean because DPT has occasional long latencies (for example, after an outage). We also consider means when there are few outliers and measure variance to judge how much outliers exist.

Finally we measure *hit percentage*, the fraction of queries for data that return data, as opposed to simply reporting no new data is available, to evaluate network overhead.

7.2 Comparing Fixed Interval and DPT

First we want to compare fixed-interval polling with our improved synchronization algorithms using data from real sensors.

Recall that in Section 4.2 and Figure 4 we shows that fixed-interval polling is difficult to do well. By default, latency is half the polling interval if the periods do not match. If periods match, then latency is governed by the relative phases, but phases require effort to keep aligned, and minor changes in phase (for example, moving just before or after the update time) causes huge changes in latency. And with a poor choice of phase, latency can be systematically bad (or good).

Figure 4 showed simulation data for an exhaustive comparison of phase and a range of periods. Turning to real deployments, Figures 14 and 15 compare fixed-interval with DPT for two different real-world sensor with a target publishing interval of 300s. For each figure, we replay the publish times of this sensor to measure use times of a consumer using fixed or each of two versions of DPT.

These examples show three things. First, real fixed-interval polling shows a range of latencies. We matched the data publishing period, but did not attempt to match phase with data publishing time as manual matching is too labor intensive. Figure 15 shows near linear latencies because this sensor was frequently restarted and each restart has a differ-

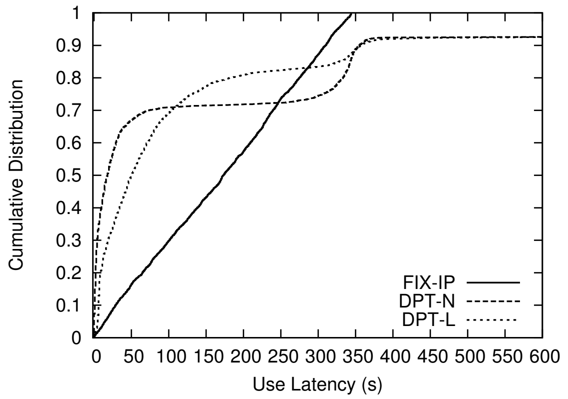


Figure 15. Use latency of all sensor publishings from one unreliable sensor (dataset: Temperature, sensor: 014-in).

ent relative phase. The sensor in Figure 14 was quite stable for weeks at a time, so latencies there show several more common values (around 40s, 140s, 200s, and 250s). These represent long periods where the consumer and the publisher run at the same relative phase.

Second, we see that DPT works very well at reducing use latency. The median latency with DPT is about one-tenth to one-third that of fixed interval for either sensor (14s vs. 232s in Figure 14, 18 or 50s vs. 150s in Figure 15). Regardless of initial setting, DPT learns to track either publisher.

Finally, we see that DPT occasionally has latency worse than fixed-interval polling. For the stable sensor (Figure 14), this happens very rarely, less than 5% of the time. For the unstable sensor, latency is worse 20–30% of the time. These large latencies are caused by DPT's backoff algorithm, when it waits up to $2^7\times$ the publishing interval. This cost is worse with an unreliable sensor where outages and large backoff is more frequent.

These figures show two representative sensors of the many we examined. We compare fixed-interval polling and DPT more systematically in Section 7.4 where we explore a wide range of failure conditions in simulation.

7.3 Comparing DPT Variants in Several Deployments

Section 7.2 compared fixed and two variants of DPT for two specific sensors.

We next turn to all sensors in each of our datasets to compare the DPT variants and fixed-interval polling in real-world conditions. We compare not only latency, but also hit percentage to study how performance and overhead trade off.

Figures 16, 17, and 18 show latency (top) and high percentage (bottom) for three deployments, Temperature, Habitat, and Seismic. We omit Location due to space limitation; its results are similar. In each case, we play back the dataset through each algorithm to evaluate latency and hit percentage.

The qualitative comparisons of these deployments are fairly similar. Across each dataset, all variants show much lower latency than fixed-interval polling. Only Seismic shows a few outliers with greater latency. This result suggests that the unreliable sensor in Figure 15 is unusual, most

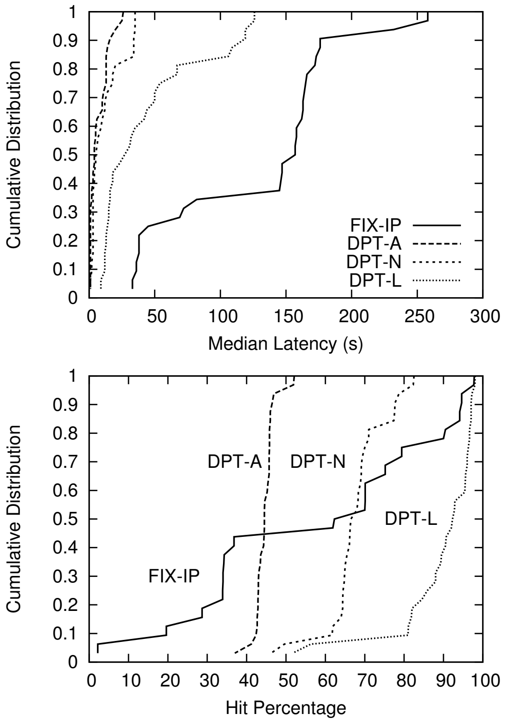


Figure 16. Performance comparison of different polling policies with traces (dataset: Temperature), latency (top) and hit percentage (bottom).

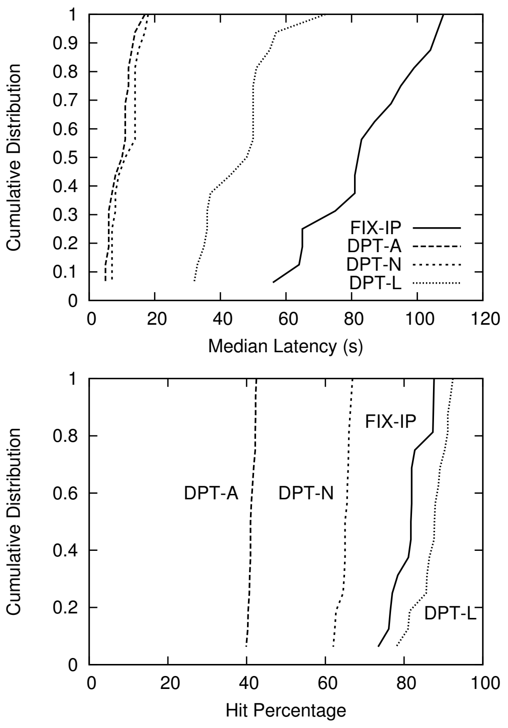
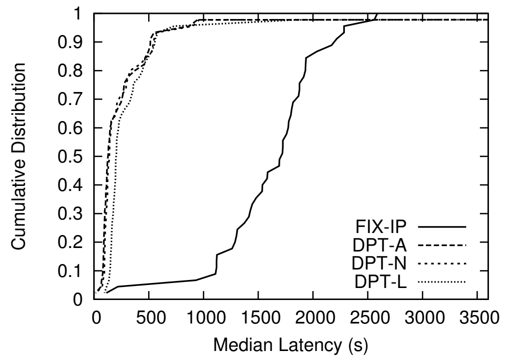


Figure 17. Performance comparison of different polling policies with traces (dataset: Habitat), latency (top) and hit percentage (bottom).



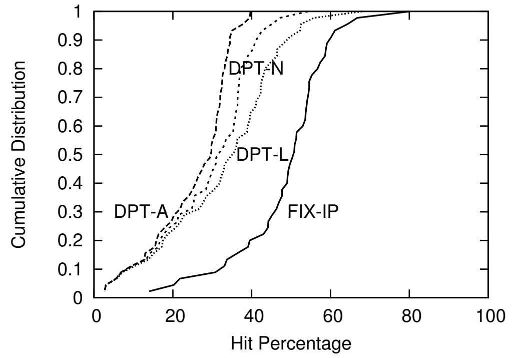


Figure 18. Performance comparison of different polling policies with trace (dataset: Seismic), latency (top) and hit percentage (bottom).

sensors in most deployments are more regular.

Comparing variants, DPT-A consistently shows the lowest latency, with DPT-N and -L usually close. Only in Habitat, does DPT-L show significantly worse latency than the others; we plan to examine this case more closely.

These figures are the first to show hit percentages, the fraction of requests that find fresh data. DPT sometimes shows better or worse hit percentages relative to fixed-interval polling, depending on the deployment. When comparing DPT variants, as predicted by Figure 13, DPT-A always has a lower hit percentage than -N and -L, although the quantitative difference depends strongly on the deployment and does not match the Laplacian prediction.

Overall we conclude from analysis of these deployments that DPT is a strict improvement over fixed-interval polling, offering greatly reduced latency with only moderate decrease in hit percentage. The DPT variant can be chosen based on preference in the latency/hit percentage trade-off. Since in most cases DPT-L provides reasonable latency and good hit percentage, we suggest that as a default.

7.4 Synchronization Performance over Many Loss Conditions

We just examined DPT performance in several deployments (Section 7.3), but those cover only four specific use scenarios. To more fully explore DPT performance under a range of loss conditions we turn to simulation with artificial and controlled traffic patterns.

Using the traffic model we describe in Section 5, Figures 19 and 20 show two "slices" through the parameter space. Each graph looks at a wide range of failure probability (governed by p_{ss}), while Figures 19 recovers more slowly after loss than Figure 20 ($p_{fs} = 0.6$ rather than $p_{fs} = 0.8$),

For each graph we ran 100 simulations for each configuration. DPT points omit confidence intervals because they were very small. For fixed-publishing we report the best and worst simulation cases as well as the mean of all simulations. Publishing interval is 300s, so the best possible case would be around 1s latency and worst would be 299s (phases aligned); the best and worst cases we report represent those of randomly chosen phases.

Evaluation of latency (the left graphs) shows that our conclusions from real-world deployments hold over this wide range of simulation parameters. In addition, when reliability is good (around $p_{ss} > 0.8$), it shows that DPT latency is as low as, or lower than the best phase of the fixed-interval cases we chose randomly.

Hit percentages (the right graphs) again vary, with DPT-L generally doing much better than typical fixed-interval polling, while DPT-A has more misses. Hit percentage of DPT-A and DPT-N are lower not only because the poll before the data is likely to be there, but then they also do 2 or 1 fast retries if they miss.

To get a better idea of consistency in results, the middle graphs show standard deviation of latency. We see that DPT-L can be quite consistent, even more than the average fixed-polling interval, when when reliability is good (around $p_{ss} > 0.8$), This observation supports our claim that DPT does particularly well with stable data sources.

From these results we conclude that DPT provides much lower latency and reasonable hit percentage for a wide range of loss characteristics.

7.5 Validity of Simulation Observations

The results of Section 7.4 are based on simulations using our traffic model, but in Section 5.2 we showed that our model is not a perfect fit for reality. For example, we do not attempt to model long outages.

To see if the known approximations of our model change the conclusions from our simulation, Figure 21 compares DPT performance from trace playback (gray bars on the right) to simulations using parameters instantiated from the same trace. The figure shows that, although there are small differences in absolute performance, our conclusions about the relative performance of the algorithms is unchanged.

We found similar results when we compare simulations and playback of other relatively stable sensors. We expect greater divergence from unstable sensors (such as that shown in Figure 15), since the model does not attempt to capture long-term outages.

7.6 Bimodal DPT and Disruption Tolerant Networking

Finally, we consider Bimodal DPT and how it interacts with deployments using Disruption Tolerant Networking. For this study we focus on the Seismic dataset since it makes the heaviest use of DTN. While some sensors in Seismic are always available, others batch reports and send them once a day for policy reasons, and others employ manual data muling and report batches of data aperiodically. Such variation

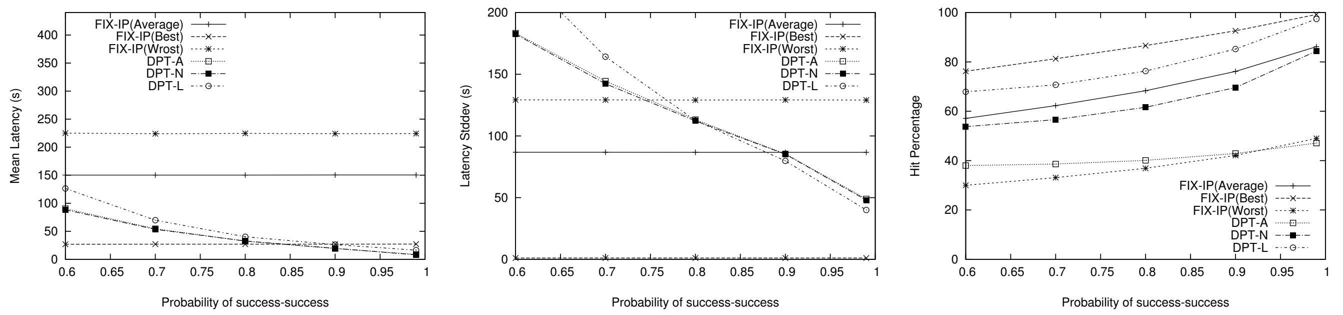


Figure 19. Performance comparison of different polling policies with moderate recovery ($p_{fs} = 0.60$): latency (left), standard deviation of latency (center) and hit percentage(right).

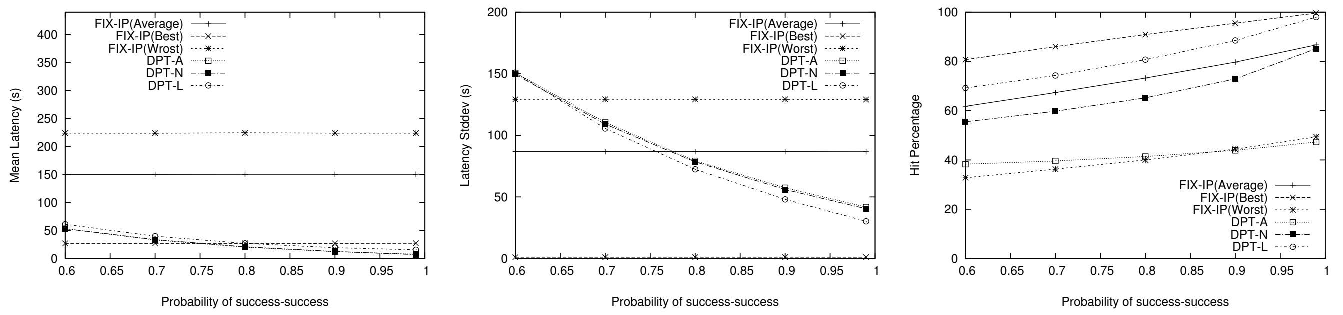


Figure 20. Performance comparison of different polling policies with faster recovery ($p_{fs} = 0.80$): latency (left), standard deviation of latency (center) and hit percentage(right).

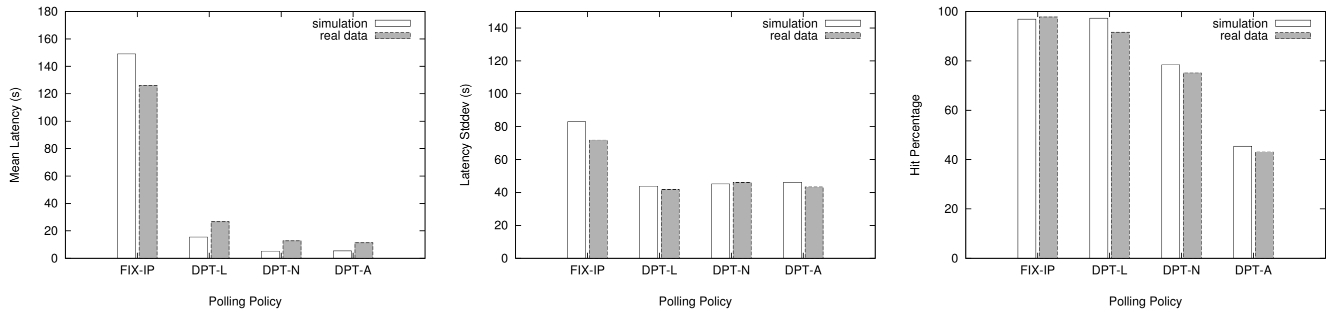


Figure 21. Comparing simulation and trace playback results for one sensor (dataset: Temperature, sensor: vir-in) for latency (left), standard deviation of latency (center), and hit percentage (right).

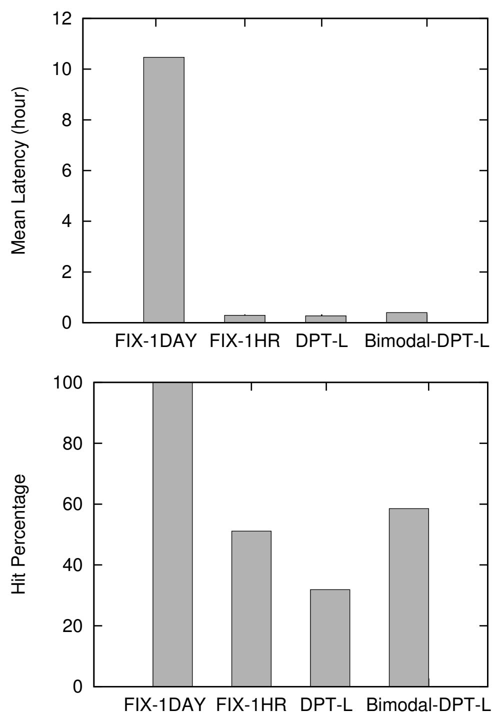


Figure 22. Comparing DPT-L with and without bimodal operation to two periods of fixed-interval polling, mean latency (top), and hit percentage (bottom). (Dataset: Seismic, sensor: PE48).

makes it difficult for DPT to track data arrival times, and motivated Bimodal DPT to try and capture both individual sensor readings and the daily patterns.

Figure 22 shows the comparison of two fixed-interval polling configurations (at 1 day and 1 hour), DPT-L with bimodal operation disabled, and Bimodal DPT-L. In this case we selected one of the six sensors that was providing batched, daily reports. In this case, fixed polling is always successful (hit percentage 100%), but latency is nearly half a day. Mean latency drops to 18 minutes with hourly polling, but the hit percentage drops to 51%.

Applying non-bimodal DPT-L to this scenario gives performance as good as hourly polling (DPT-L mean latency is 16 minutes), but with an even lower hit percentage. This problem occurs because DPT trains on the short interval when data is appearing during connectivity, but then it suffers many misses as it backs off during the day.

Bimodal-DPT-L, by comparison, learns both periods and so provides about the same latency (mean of 24 minutes), but with a much better hit percentage (59%). Bimodal-DPT still will suffer several misses while it times out during a daily outage, but it knows to take a long (bimodal) pause at that point.

We conclude that bimodal-DPT is important if both good hit percentage and low latency are desired.

7.7 Multi-source DPT

Lastly, we evaluate the multi-source synchronization algorithm we introduced in Section 6.5. Similar to the single source synchronization, we define the republishing latency as the time that elapses since the republishing condition is satisfied and the republishing hit percentage as the fraction of successful republishing that satisfy the republishing condition. When multiple sources are used in a republishing, tracking individual source is not sufficient to reduce the republishing latency. It is necessary to coordinate the sources and initiate the republishing when the republishing condition is satisfied. Here we show our multi-source DPT algorithm manage multiple sources and reduce the republishing latency by coordinating them.

We consider following multi-source republishing scenario to evaluate the performance of our multi-source DPT. First, we generate three sources using our publishing model which has different periods and phases. One starts at 0s with period of 500s. The others have a longer period of one hour, but they have different phases. They have initial phases of 20 and 40 minutes, respectively, relative to the first source.

We set the republishing condition as that the republishing initiate only if two or more of sources have updates. With the various intervals and phases of sources, it is difficult to achieve both low latency and high hit percentage with a fixed republishing interval.

In this scenario we compare the performance of MDPT to that of fixed interval polling shown in Figure 23. We tested fixed polling with three different intervals that are 60, 30, and 15 minutes. The intervals are chosen to be close to the publishing periods of sources. For fixed interval polling we see a clear trade-off of latency and hit percentage according the fixed interval. The latency becomes about half of the polling interval. The hit percentage decreases as the polling interval become short. Although 15 minutes polling interval gives very reasonable latency of 7.8 minutes, it gets only 47% of successful queries. Compared to this fixed-interval polling, our algorithm achieves the lowest latency of 48s, yet the hit percentage is equal or higher than any simple period polling (96%). This shows that a smart polling can achieves low latency with much less polling than that fixed interval polling requires.

We also evaluate MDPT with the real trace that includes phase shifts, interval variation, and outages. All source has approximately 5 minutes period but each has a different phase. We test three fixed interval polling (10, 5, 3 minutes) with random initial phase which makes the mean latency varies. Figure 24 shows the performance of MDPT and fixed interval polling. The latency of MDPT is 1.5 minutes which is lower than the latency with 3 minutes interval polling (the lowest latency among fixed-interval polling), yet the hit percentage is 89%, close to 10 minutes fixed-interval polling (the highest hit percentage among fixed-interval polling). Compared to fixed interval republishing, our approach reduces the republishing latency and increase the republishing hit percentage without manual tuning of interval or phase.

Overall, our MDPT coordinates the individual sources and predicts the time when republishing condition becomes satisfied, which generally better performance than fixed interval republishing.

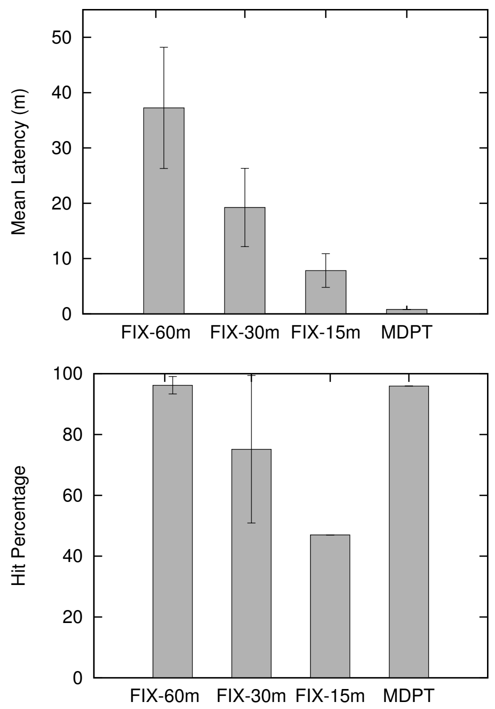


Figure 23. Comparing MDPT to fixed interval pollings, mean latency (top), and hit percentage (bottom). (Publishing Model with Pss=0.95 and Pfs=0.80).

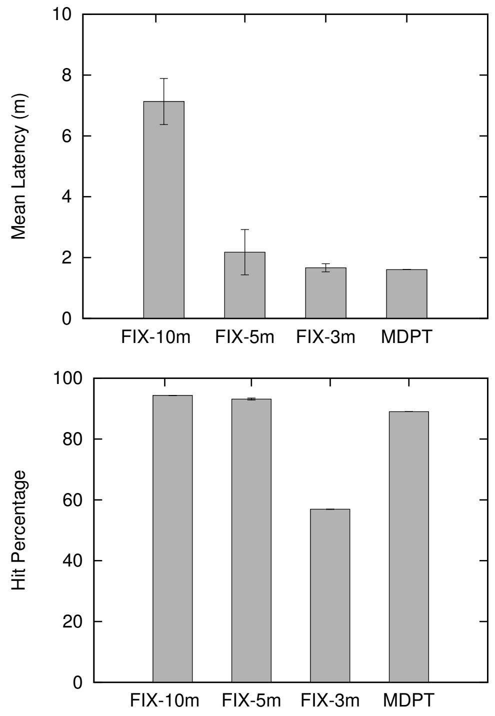


Figure 24. Comparing MDPT to fixed interval pollings, mean latency (top), and hit percentage (bottom). (Dataset: temperature. sensors: vir-out, bay-st-out, sum-out).

8 Conclusions

In this paper we described the problem of synchronizing sensor publishers and consumers. While sensornets have traditionally generated data and consumed it by polling at fixed intervals, that approach adds significant latency for each processing step. To promote a rich environment with many people generating, consuming, and republishing data, we introduced Data Publication Tracking, an approach that allows data consumers to synchronize efficiently with multiple publishers.

Imposing very little cost on publishers, we showed that this approach reduces median latency in each processing step to 10-30% of what fixed-interval polling would require in four real-world deployments.

Acknowledgment

This work is supported by National Science Foundation (NSF) grants CNS-0626702, Sensor-Internet Sharing and Search.

We thank Deborah Estrin and Martin Lukac (UCLA) and Fabio Silva (USC/ISI) for providing access to three of our datasets and answering questions about their deployments.

We thank Mark Hansen and Junghoo Cho for the discussions that helped motivate this work.

9 References

[1] Karl Aberer, Manfred Hauswirth, and Ali Salehi. A middleware for fast and flexible sensor network deployment. In *VLDB*, pages 1199–1202, 2006.

[2] Kevin Chang, Nathan Yau, Mark Hansen, and Deborah Estrin. Sensorbase.org - a centralized repository to slog sensor network data. 2006.

[3] M. Colagrosso, W. Simmons, and M. Graham. Demo abstract: Simple sensor syndiciation. In *Proceedings of the Fourth ACM SenSys Conference*, pages 377–378, Boulder, Colorado, USA, November 2006. ACM.

[4] Edward J. Daniel, Christopher M. White, and Keith A. Teague. An inter-arrival delay jitter model using multi-structure network delay characteristics for packet networks. In *the 37th Asilomar Conference of Signals, Systems, and Computers*, volume 2, pages 1738–1743, November 2003.

[5] Chad Dickerson. Rss growing pains. http://www.infoworld.com/-article/04/07/16/29OPconnection_1.html, July 2004.

[6] Robert F. Dickerson, Jiakang Lu, Jian Lu, and Kamin Whitehouse. Stream feeds - an abstraction for the world wide sensor web. In *IOT*, pages 360–375, 2008.

[7] Kevin Fall and Stephen Farrell. DTN: An architectural retrospective. 26(5):828–837, June 2008.

[8] Cathy A. Fulton and San qi Li. Delay jitter first-order and second-order statistical functions of general traffic on high-speed multimedia networks. *IEEE/ACM Transactions on Networking*, 6(2), April 1998.

[9] Phillip B. Gibbons, Brad Karp, Yan Ke, Suman Nath, and Srinivasan Seshan. Irisnet: An architecture for a worldwide sensor web. *IEEE Pervasive Computing*, 02(4):22–33, 2003.

[10] J. Gray, W. Chong, T. Barclay, A. Szalay, and J. Vandenberg. Terascale sneakernet: Using inexpensive disks for backup, archiving, and data exchange, 2002.

[11] Jonathan W. Hui and David E. Culler. Ip is dead, long live ip for wireless sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 15–28, New York, NY, USA, 2008. ACM.

[12] Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Allen K. Miu, Eugene Shih, Hari Balakrishnan, and Samuel

Madden. CarTel: A Distributed Mobile Sensor Computing System. In *4th ACM SenSys*, Boulder, CO, November 2006.

[13] Allen Husker, Igor Stubailo, Martin Lukac, Vinayak Naik, Richard Guy, Paul Davis, and Deborah Estrin. Wilson: The wirelessly linked seismological network and its application in the middle american subduction experiment. May/June 2008.

[14] The Weather Underground Inc. Weather Underground. http://-wunderground.com, 2006.

[15] Aman Kansal, Suman Nath, Jie Liu, and Feng Zhao. SenseWeb: An infrastructure for shared sensing. 14(4):8–13, October 2007.

[16] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, 2006.

[17] Martin Lukac, Lewis Girod, and Deborah Estrin. Disruption tolerant shell. In *CHANTS '06:Proceedings of the SIGCOMM workshop on Challenged networks*, pages 189–196, Pisa, Italy, September 2006. ACM.

[18] Suman Nath, Amol Deshpande, Yan Ke, Phillip B. Gibbons, Brad Karp, and Srinivasan Seshan. IrisNet: An architecture for internet-scale sensing services.

[19] Suman Nath, Jie Liu, and Feng Zhao. Challenges in building a portal for sensors world-wide. In *First Workshop on World-Sensor-Web*, Boulder,CO, October 2006. ACM.

[20] Unkyu Park and John Heidemann. Provenance in sensornet republishing. In *Provenance and Annotation of Data and Processes: Second International Provenance and Annotation Workshop, IPAW 2008*, pages 280–292, Salt Lake City, Utah, USA, June 2008. Springer Verlag.

[21] Sasank Reddy, Gong Chen, Brian Fulkerson, Sung Jin Kim, Unkyu Park, Nathan Yau, Junghoo Cho, and John Heidemann Mark Hansen. Sensor-internet share and search—enabling collaboration of citizen scientists. In *Proceedings of the ACM Workshop on Data Sharing and Interoperability on the World-wide Sensor Web*, pages 11–16, Cambridge, Mass., USA, April 2007. ACM.

[22] Kepler project. http://kepler-project.org/.

[23] Ka Cheung Sia, Junghoo Cho, and Hyun-Kyu Cho. Efficient monitoring algorithm for fast news alerts. *IEEE Transactions on Knowledge and Data Engineering*, 19(7):950–961, 2007.

[24] Robert Szewczyk, Alan Mainwaring, Joseph Polastre, John Anderson, and David Culler. An analysis of a large scale habitat monitoring application. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 214–226, New York, NY, USA, 2004. ACM.

[25] Igor Talzi, Andreas Hasler, Stephan Gruber, and Christian Tschudin. Permasense: investigating permafrost with a wsn in the swiss alps. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, pages 8–12, New York, NY, USA, 2007. ACM.

[26] Cornell University. The Great Backyard Bird Count. http://audubon.-org/gbbc, 2006.

[27] Geoffrey Werner-Allen, Konrad Lorincz, Matt Welsh, Omar Marcillo, Jeff Johnson, Mario Ruiz, and Jonathan Lees. Deploying a wireless sensor network on an active volcano. *IEEE Internet Computing*, 10(2):18–25, 2006.

[28] Dave Winer. RSS 2.0 Specification. http://blogs.law.harvard.-edu/tech/rss, 2002.

[29] A. Woo, S. Seth, T. Olson, J. Liu, and F. Zhao. A spreadsheet approach to programming and managing sensor networks. *Proc. of the Fifth Int. Conf. on Info. Processing in Sensor Networks*, pages 424–431, 2006.

[30] Yahoo. Yahoo Pipes. http://pipes.yahoo.com/pipes/.

[31] Tomi Yletyinen and Raimo Kantola. Voice packet interarrival jitter over ip switching. In *SBT/IEEE International Telecom Symposium ITS '98*, volume 1, pages 16–21, 1998.

[32] Li Zheng, Liren Zhang, and Dong Xu. Characteristics of network delay and delay jitter and its effect on voice over ip (voip). In *IEEE International Conference on Communicaitons ICC*, volume 1, pages 122–126, 2001.