Tinker: A Tool for Designing Data-Centric Sensor Networks

Jeremy Elson
Microsoft Research
One Microsoft Way
Redmond, WA 98052
jelson@microsoft.com

Andrew Parker
Center for Embedded Networked Sensing, UCLA
420 Westwood Boulevard
Los Angeles, CA 90095
adparker@gmail.com

ABSTRACT

We describe *Tinker*, a high-level design tool that aids the exploration of the design space in sensor network applications. Tinker is targeted at applications that require real-time assignment of semantic meaning to data, rather than just data storage. Tinker lets users write simple programs, as if they were manipulating individual scalar values, and simulates those computations over continuous streams of sensor data. Tinker does not require (or allow) users to specify details such as routing algorithms or retransmission policies, freeing system designers to rapidly iterate among different broad designs before fleshing out details of the one that looks most promising. We demonstrate Tinker's use in the design and deployment of *ElevatorNet*, our distributed sensor application that retrofits buildings with per-floor displays of an elevator's position, determined using barometric altimetry.

Categories and Subject Descriptors

D.2.2 [**Software**]: Software Engineering—*Design Tools and Techniques*

General Terms

Design, Experimentation

Keywords

Sensor networks, design tools

1. INTRODUCTION

There has been an explosion of interest in sensor networks, yet there is still a relative shortage of sensor network applications that use collected data *in situ*. In other words, the primary design goal in many sensor networks is data *transport*, not data *interpretation*. Systems designed to record data for later study are valuable, but they seem to fall short of the vision held by many sensor networks researchers: early position papers [2, 3] dreamed of systems that were constantly *reacting* to their environment, not just recording it. Data collection systems treat sensor data only as a packet payload. Absent any requirement to assign semantic meaning to data *in situ*, a sensor network is essentially a traditional ad-hoc wireless network, albeit one with significant energy constraints and with a high ratio of nodes to human maintainers.

Of course, interesting systems have been built that perform *in situ* data interpretation [13, 15, 1, 5]). Those that do physical actuation in response to stimuli are perhaps the best examples (e.g., [6, 18]). But, such applications seem (to us) in the minority. Our interest is piqued by applications that close the loop of sensing, interpretation, and actuation; we'd like to create tools that make those applications easier to build.

In this paper, we describe *Tinker*, a tool for exploring the design space of sensor network applications that assign semantic meaning to the data they collect. That is, given a trace of real sensor inputs and a desired output, Tinker lets designers explore different algorithms to find one that works best. "Best" has an application-specific meaning, defined by the designer—perhaps the most resilient to outliers, the least degraded by missing data, or the fastest to respond after a change.

Tinker is not a simulator in the sense of TOSSim [10], EmStar [4], or Avrora [17]. Such simulators are crucial for working out protocol details, finding implementation bugs, and so forth. Tinker, in contrast, is a higher level design tool meant for use *before* code-writing begins. It encourages designers think more abstractly about the steps required to transform input into useful output, and rapidly iterate through design alternatives without the burden of working through the implementation details of each one. Real-code simulators are commonly used to answer questions like "How many packets were lost due to transient routing loops?" or "How many collisions were avoided using this new MAC layer?". Tinker is meant to answer questions like "Is it better to transmit every time-series value back and average it at the base station, or transmit exponentially weighted moving averages computed locally on the node?" Tinker is complementary to real-code simulators, not a replacement.

The remainder of this paper is organized as follows. Section 2 describes Tinker and demonstrates some of its features through simple examples. Section 3 describes *ElevatorNet*, a real system we built with the help of Tinker. ElevatorNet can report the position of a building's elevator in real-time using barometric altimetry. Its design was non-obvious, and Tinker-based simulations were vital in finding the right algorithm to reliably convert the output of pressure sensors to a floor number. In Section 4 we review related work, and give our conclusions in Section 5.

IPSN'06, April 19–21, 2006, Nashville, Tennessee, USA.

2. TINKER

Tinker is a collection of Python [11] modules that create an imperative language for simulating the computations to be performed over values in a data stream as they arrive. It is also a library of simple operators that are commonly used in sensor network data processing: algebraic operators, tracking of minimum or maximum values in a series, computed exponentially weighted moving averages, and so forth. For example, the following code tracks the difference between the most recent value and the historical maximum:

```
p = compute.ColumnName(dataset, "Pressure")
max = compute.Max(dataset, p);
diff = compute.Sub(dataset, p, max);
```

The code above reads like a computation on scalar values: we first pull a value out of a dataset and assign it to `p`, then call the functions `Max` and `Sub` on the returned values. However, the values being computed are not scalar; the code above actually describes a series of computations executed each time new data arrives. `p` is not a single pressure reading, but an array of them. Similarly, the local variables created (e.g., `max` and `diff`) do not contain scalar values, but automatically-generated references to an array that grows to contain all the output values. (Since Tinker is a Python library, the normal Python rules of syntax and scope apply. For example, the local variables `max` and `diff` are created implicitly when they are used.)

An easy way to visualize the data flow is as a spreadsheet: both the inputs (`p`) and the outputs (`max`, `diff`) are column names, and each row represents a time step. Tinker requires each computational step to take one (or a few) scalar values as inputs, and generate a scalar value as output.

New Tinker operators are easy to write. For example, the `Max` operator looks like this:

```
class Max(Computer):
  def __init__(self, data, target):
    self.max = None;
    self.result=data.map_one_col(target, self);

  def func(self, value):
    if self.max == None or self.max < value:
      self.max = value;
    return self.max;
```

User-supplied operators like `Max` are all derived from Tinker's base `Computer` class. Each must supply an initialization function `__init__` and the computation itself, in the `func` function. In this example, the initialization function takes three arguments. The first two (`self` and `data`) are housekeeping variables common to all operators. `target` is the operand—the name of the column of input data over which a maximum should be computed. Operators can, optionally, be stateful, as in the example above: the local variable `self.max` is initialized to `None` when the operator `Max` is instantiated, and referenced later when data values are issued. Tinker operators can also take any number of parameters when they are instantiated.

In Tinker, any operator's `func` is called once for each row, with that row's value from the `target` column passed as an argument. In the case of the `Max` operator, the function compares the most recently issued `value` with its internal state `self.max`. If the maximum value is undefined (i.e., this is the first value in the column to be seen), or if the current value is larger than the previous maximum, the maximum is updated. The Tinker framework saves each intermediate value for analysis, described in more detail in Section 2.1.

2.1 Input and Output

Tinker was written with a general application design strategy in mind. The assignment of semantic meaning to data can not be done before understanding the nature of both the sensors and their target environment. To gain this understanding, designers are expected to instrument that environment, collect representative data annotated with ground truth, test an analysis technique using Tinker, then iteratively refine both their analysis and instrumentation strategies. We will show an example of this cycle in Section 3.

Tinker does not address how data is collected. Users are expected to collect data in whatever way is convenient given the nature of their deployment. Tinker only assumes that data is written to text files in a simple comma-separated-value (CSV) format, one time step per line, and one datum per column. The first row is assumed to be special, containing column titles that are later referenced within Tinker. The following is a simple file that might be given to Tinker as input:

```
time,seqno,temp,pres
200,1001,26.2,1000.04
400,1002,26.2,1000.09
600,1003,26.2,1000.05
800,1004,26.2,1000.18
```

This format is easy to generate and parse. Within Tinker, data files are first read by instantiating the Data operator, taking the name of a data file as a parameter. Subsequently, columns are referenced using the `ColumnName` operator, which takes a string parameter that is expected to match one of the column header strings. Finally, `output` is called to generate output, again using a filename as a parameter.[1] For example:

```
dataset = compute.Data("my_data_file.txt");
p = compute.ColumnName(dataset, "pres")
p_smooth = compute.EWMA(dataset, p, 30);
p_max = compute.Max(dataset, p_smooth);
dataset.output("processed_data.txt");
```

After all computation has been performed, Tinker emits a CSV-formatted output file similar to the input file. The output contains all of the original input data, plus extra columns that contain the intermediate state of each computational operator after each timestep. The output data is also annotated with extra information that makes it easy to visualize using GnuPlot and a Tinker helper script. Since the output is a simple CSV-formatted text file, it can also be easily imported into tools such as Excel and Matlab for analysis or visualization.

There are two ways to name the columns that contain the output and side-effect of computations. The first option is an "auto-generated" column name. Operators can be given optional `name` callback functions that generate new output column names based on the input column names and parameters. For example, Tinker's EWMA (Exponentially Weighted Moving Average) operator contains the following function:

```
def name(self, target):
  return "ewma%d(%s)" % (self.max_w, target);
```

Output columns are given automatic names by passing the input column name to the operator's `name` function. For example, in the Tinker program shown earlier in this section, the output datafile will have two extra columns appended—one for the call to EWMA, and one for the call to Max. The EWMA output column will have an

[1] Omitting the filenames causes Tinker to use `stdin` or `stdout`.

automatically generated default name `ewma30(pres)`. This name is the result of Tinker passing the `pres` column name to EWMA operator instance created with a max_w=30 parameter.

Automatically generated names are useful, but can become cumbersome after several chained computations. Tinker also provides the `SetName` function for giving a specific name to an output column. This is useful for giving a semantic title to a result, rather than explicitly describing the series of operators that were applied to compute it. For example, we can add one extra line to the Tinker program above:

```
p_max.SetName("HighestPressure");
```

The call to `SetName` replaces the automatically generated name `Max(ewma30(pres))` with `HighestPressure`.

2.2 Energy Conservation and Data Loss

An important class of Tinker operator is its *loss* operators. These operators are used to study the effects of missing or incomplete data. There are two principal reasons that incomplete datasets are common in sensor networks. First, packet loss is frequent in low-power wireless networks. In some applications, retransmission is costly, so it is impractical to sweep packet loss under a retransmission rug. Second, packet "loss" is sometimes intentional. That is, to conserve bandwidth, various forms of filtering (e.g., thresholding, event detection) can be used at the sensor to suppress the transmission of uninteresting data.

In both cases, the effect is similar: the final result, when computed over incomplete input, may be different than the result with all input available. The magnitude of the difference, of course, is heavily application dependent. Some algorithms are sensitive to loss, while others are resilient—using techniques such as averaging over multiple samples, waiting for consensus before taking action, and so forth. Tinker is useful precisely in that it can help designers understand how *their* applications respond to loss, and tinker with various algorithms to see which is the most resilient.

For example, consider a sensor that is producing a continuous stream of pressure data, and the user wants a display that shows the best estimate of pressure, averaged over the past minute. There are at least three possible algorithms, with various trade-offs:

1. Transmit every sample back to the base station. The base station computes the average.
2. Transmit a sample back to the base station every time it varies from the previous sample by more than x units. The base station assumes all the missing data has the value of the most-recently-heard sample.
3. The sensor keeps a running average computed locally; every sampling period, it transmits the new average.

None of these algorithms is the best in every situation—they have advantages and disadvantages along axes such as response time, total bandwidth utilization, and sensitivity to loss.

Tinker makes it easy to analyze each of these hypothetical situations. Consider the following `Threshold` operator:

```
class Threshold(compute.Computer):
  def __init__(self, data, target, thresh)
    self.last_value = None;
    self.thresh = thresh;
    self.rx = self.tx = 0;
    self.result=data.map_one_col(target, self);
    sys.stderr.write("Duty cycle: %.4f" % \
                     (100.0*self.tx/self.rx));

  def func(self, value):
    self.rx += 1;

    if self.last_value==None or self.thresh <=
          abs(self.last_value - float(value)):
        self.tx += 1;
        self.last_value = value;
    return self.last_value;
```

This operator simulates Algorithm 2, with a configurable threshold passed as an argument to the operator. After Tinker runs, the operator reports how many packets it transmitted. The base station's view of the data at each time step is recorded in the output column generated by the operator. Using other operators, the missing-input view of the output can be easily compared to the full-input version, allowing designers to quickly gauge the effect of various filtering policies on the output as seen by the user. We will see a concrete example of this in Section 3.4.

Lossy channels can be similarly evaluated. Of course, because Tinker is a high-level simulator, there is no channel model. Tinker is the wrong tool, for example, to evaluate a new media access control protocol. Tinker *is* well suited for quickly trying out different data processing strategies to learn which is most resilient to loss.

2.3 Tinker as a Design Tool

We said in Section 1 that our goal was to focus on sensor networks that assign semantic meaning to data instead of just storing it. We think Tinker helps with these applications for several reasons.

First, Tinker encourages a design cycle where the first consideration is on the *data*. When building a new sensor application, it is easy to get sucked into systems details too early: retransmission protocols, routing algorithms, and so forth. The data is an afterthought—just a packet payload!

Second, Tinker's programming model "enforces" (strongly encourages) data processing in *real time*. Tinker does not give operators reference to entire input arrays. Instead, it issues input data to operators one datum at a time, and forces operators to issue their corresponding output after every time step. This makes it impossible, for example, to smooth data using a sliding window centered at the current time step: the operator can't see into the future. Yet, under this programming model, the exponentially weighted moving average is easy to write. The distinction is subtle, but important: a sliding-window average can't be used in a "live" system, but only for *post-facto* analysis. Tinker's programming model encourages designers to stay in the live-analysis mindset.

Finally, Tinker encourages designers to think about the data flow of their system at a high level, before implementation starts. Many researchers (including this paper's authors) have advocated real-code simulators because they ease the transition from simulation to deployment. Unfortunately, this had the unintended consequence of making code-writing start too soon. Designers should play in the design space before working out every detail of one design. This argument is similar to Lamport's strong endorsement of formal specification [7]. Of course, Tinker is not a formal specification language, but, like TLA+, it does take a step toward divorcing design from implementation.

To make Tinker's use as a design tool more concrete, we now describe a real application built with the help of Tinker: *ElevatorNet*.

3. ELEVATORNET

We first developed and tested Tinker in the context of an application we call *ElevatorNet*. ElevatorNet started as an easy way to fix a pet peeve. Our seven-floor building has three elevators. Most floors do not have displays that show where the elevators are. This can be frustrating to people waiting. It also causes delays: without knowing which of the independently-signaled elevators is closer, riders push both "call" buttons and take the first elevator to arrive. To add floor displays in the traditional way, our building's management quoted a labor and materials cost of about $50,000 ($2,300 per elevator per floor).

Our goal was to create a simple, non-invasive system that would be cheaper and easier to install, but have the same functionality. A wireless sensor network seemed a good match to this problem. Our office has a wireless network infrastructure (802.11), but ElevatorNet could not use it: base-stations are not reachable from the three parking garage levels, or from inside the elevator when the doors are closed. 802.11-based devices would also have a shorter lifetime since they require more power (at least, in this low-data-rate application, where the economy of scale of 802.11 is not realized). Line power is not easily available inside our elevators.

We decided instead to build a prototype using an ad-hoc network of motes, both inside the elevator to sense its position, and on each floor to for peer-to-peer relaying of position data throughout the building. The entire system is installable in 30 minutes with no tools other than Velcro.

Perhaps the most important unknown in our initial design was the method for sensing elevator position. We considered various techniques, such as sensing motion with accelerometers or using radio signal strength, and finally decided to prototype a system using barometric altimeters. A barometric altimeter determines its height from a reference plane by measuring air pressure. Many common altimeters (such as those found in some GPS devices, or in general aviation aircraft) can sense vertical motions of about 5 to 10 feet. However, the relationship between pressure and true altitude changes along with environmental conditions, such as temperature and ambient air pressure. Temperature and pressure are driven by local weather; indicated altitude at a fixed point will vary by 20 to 30 feet (two or three floors) over the course of a typical day.

ElevatorNet was also interesting to us because it required real-time assignment of semantic meaning to a stream of data. Our goal was not to store altimeter readings in a database, but to keep an elevator floor display updated in real-time. We used Tinker to tinker with an algorithm until we found one that was responsive and robust.

3.1 Sensor Analysis

The first stage of our project was a feasibility study: can a cheap electronic altimeter sense altitude with enough precision to differentiate floors of a building, without being confounded by environmental variations? To answer this question, we had to characterize both the sensors and the environment they'd be sensing.

ElevatorNet used the Intersema MS5534, a piezoresistive pressure sensor with a built-in analog-to-digital converter and a simple 3-wire interface. We used Moteiv's "Tmote Sky" wireless sensors (descendants of the Berkeley Telos B mote). Moteiv also provided an interface board between the MS5534 and their Tmote Sky.

The sensors have a specified pressure resolution of 0.1 millibars, which translates to about 2.7 feet of altitude near sea level. After applying the calibration data provided by the factory with each sensor, we found the typical precision of an individual, stationary sensor at a constant temperature to be about $\pm$0.1mbar, as specified. However, the absolute accuracy of the sensors is considerably worse: $\pm$1.5mbar, or about 40 feet (four building floors). We then analyzed 48 hours of observations made by pairs of sensors in close proximity. This revealed that while each sensor has a large bias, the bias seemed to be constant with respect to both time and absolute pressure.

These observations led us to try a simple (but, as we will see, flawed) algorithm for floor detection. First, we subtracted the barometric pressure inside the elevator from the pressure measured at a fixed point inside the building. We expected this would give us an altitude output independent of weather conditions. To make the system easier to install, we tried using an auto-calibration algorithm to deduce the sensor biases rather than calibrating them manually. Our auto-calibration method started out simple: the system tracked the minimum and maximum altitudes seen (after the correction for ambient pressure changes), and assumed each floor of the building was 1/6th of that span.

3.2 Using Tinker to Test our Hypothesis

At this point in our project, we were not yet concerned with the implementation details: the choice of routing algorithm, for example. We still needed to test and refine our basic understanding of the application space. How accurate was our floor-detection algorithm? How robust was it to data loss and environmental variations? How responsive was it?

To answer these questions, we used the general procedure: 1, collect data; 2, try various algorithms, using Tinker; 3, redesign the system and return to Step 1.

Data collection. We collected pressure data from inside the moving elevator while manually annotating it with ground truth data about the elevator's actual location. For annotation, we wrote a simple program that allows single-keystroke entry of the elevator's position, and time-synchronized it with the pressure data. (The hard part was explaining why a researcher was sitting alone in an elevator for two hours.) We concurrently collected pressure data from a fixed point in the building: an office about 100 feet from the elevator lobby.

Tinkering. We used Tinker to test many variations on processing the pressure data into a floor number, and its GnuPlot helper script to visualize the results. This process was crucial: it let us iterate through many variations of the algorithm and test them on a substantial (two-hour) dataset annotated with ground truth. Tinker's goal of letting us think abstractly about the problem before starting to write code to deal with the implementation details saved substantial time. Furthermore, it enabled us to focus on the data rather than the system. A typical analysis is shown in Figure 1.

Redesign. Not all variations on the system can be explored using simulation alone, of course. For example, after our first round of tinkering, we discovered the origin of many errors: the long-term environmental variations in the elevator did not seem to match those in the office where our fixed-position pressure reference point was located. We collected a second annotated data set, this time with two pressure references: one at the top-floor elevator lobby, and one at the bottom. We hypothesized that these locations would be better coupled to the elevator than the office, which was further away.

After several iterations, we arrived at the final algorithm, described in the next section.

3.3 Performance of the Final Algorithm

After tinkering with various alternatives, the final algorithm we used was somewhat complicated:

- Each elevator has a single pressure sensor sampling at 5Hz.

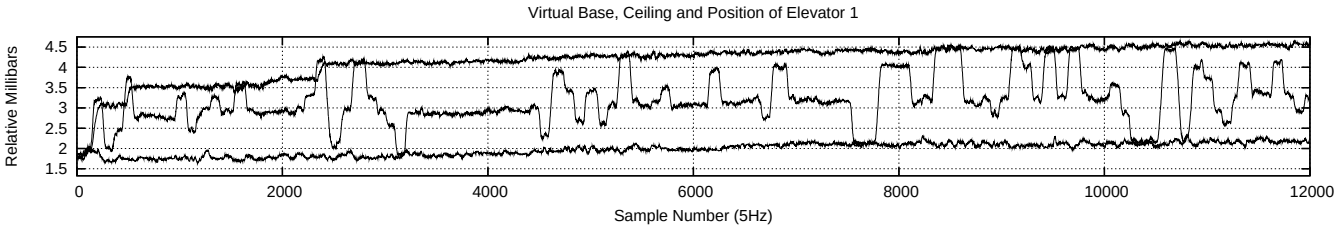

Figure 1: A recording of one elevator's position for about 40 minutes. The elevator first visits the top floor at about $t = 2500$, after which ElevatorNet correctly deduces the position of the elevator with over 99% accuracy using the algorithm described in Section 3.3. The gradual upward trend is due to changes in the weather.

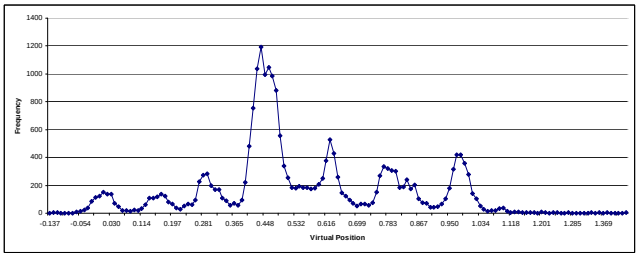

Figure 2: A histogram of the Virtual Position values seen during a 75-minute calibration test of ElevatorNet. The clear peaks in the histogram correspond to each of the 7 floors of the building, with larger peaks corresponding to floors more commonly visited. The center peak is the building lobby; peaks to the left are parking garage levels, while peaks to the right are office levels.

Pressure data was also collected at 5Hz from two reference points, one each at the top- and bottom-floor elevator lobbies.

- The *Bottom Offset* is computed at each time step. In essence, this is the difference of two sensors' calibration biases: the elevator sensor and the bottom-floor fixed sensor. It is computed as the minimum difference seen between those two sensors, after their outputs are smoothed using slow-response (10-second) exponentially weighted moving averages. It produces a correct bias estimate after the elevator first visits the bottom floor of the building. An analogous procedure is used to compute the *Top Offset*, which becomes correct after the elevator's first visit to the top floor.

- The Top and Bottom Offsets, respectively, are added to the most recent value of the top and bottom sensors. A fast-response (1-second) EWMA is applied. The resulting values are the *Virtual Base* and *Virtual Ceiling* of the elevator shaft—the expected value of the elevator's sensor when it's at the top or bottom of the shaft.

- The *Virtual Position* of the elevator is computed as a fraction of the distance from the Virtual Base to Virtual Ceiling. This is a number from 0 to 1. 0 is the current estimated pressure at the bottom of the shaft; 1 is the estimate at the top. The Virtual Position is valuable because it is a position measurement that is highly resilient to changes in ambient barometric pressure and the (considerable) individual biases of the pressure sensors. Figure 1 shows an example of the elevator's Virtual Position captured from the real system.

- When the system is first set up and activated, the installer takes the elevator to each floor of the building for 30 seconds.

Filter Threshold	Display Accuracy	Rel. Duty Cycle	Error Duration (msec)	
			Average	Longest
0	99.57	1.000	336	2000
1	99.57	0.577	368	2000
2	98.77	0.169	862	8200
3	94.34	0.046	1492	16200
4	84.90	0.018	2386	26800
5	76.48	0.012	3540	64600

Figure 3: Elevator floor detection accuracy vs. radio utilization for various thresholding values, described in Sections 3.3 and 3.4.

A histogram of the Virtual Positions visited during this setup phase shows each floor clearly, as seen in Figure 2. A Virtual Position Table is constructed which maps the center value of each peak to a floor name. (This was done manually in the prototype; automatic peak-finding is planned.)

- After this labeling step, the system becomes active. As the elevator moves, its Virtual Position is compared to the entries in the histogram-derived Virtual Position Table; the label of closest match is displayed.

This algorithm was not obvious, or our first attempt. It was the result of several weeks' iterative tinkering and data collection. Our first (incorrect) algorithm divided the Virtual Position space evenly into equally-spaced floors. Tinkering revealed that our building's floors are not exactly the same height, and the nonlinearity between altitude and pressure was unexpectedly observable even over our building's short (70-foot) height. The histogram approach proved quite robust once we found the right set of environmental corrections that would ensure the values seen during calibration would match the values seen later, during system operation.

We evaluated ElevatorNet's performance using a pressure dataset that was manually annotated with ground truth. We define the accuracy as the percentage of the time that ElevatorNet's display matches the real position of the elevator as recorded by our manual annotations. Our results exclude the timesteps where our ground truth indicated the elevator was in a transition between floors. We also excluded the initial calibration phase (i.e., when the elevator had not yet visited every floor). The remaining data span 66.3 minutes during which the elevator moved 114 times. ElevatorNet computed the floor correctly 99.57% of the time—a cumulative error of 17 seconds.

We also characterized the typical duration of an error. The granularity of our output is 200msec (the sampling rate). Most errors were transient—an incorrect result visible for 200msec and replaced by the correct result at the next timestep. The average duration of an error was 336msec. The longest single error event was 2000msec.

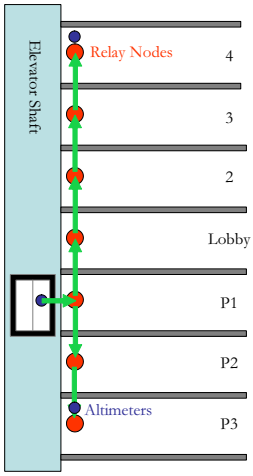

Figure 4: ElevatorNet system diagram. Each floor and each elevator was equipped with a mote. Altimeter-equipped motes in the elevators tracked their movement. Motes at the top and bottom of the building provided an ambient pressure reference. Data from all altimeters were broadcast using simple flooding, which worked well because the topology was (almost) linear.

3.4 Energy Savings with Tinker

While ElevatorNet's first prototype worked well, it did not attempt to conserve energy: every sample was transmitted to every display. The elevator is usually stationary, so many samples are redundant. Ideally, we would like local thresholding at each sensor, only issue a new reading when the elevator moves.

Unfortunately, "redundant" data is not completely redundant. ElevatorNet uses pressure sensors on the edge of their intended resolution. They are noisy; the output wanders over several feet per second. Our floor detection algorithm only worked after several layers of smoothing were applied. The environment is also noisy, on several timescales; much of the noise can not be filtered out until the elevator pressure is merged with the fixed-point reference pressure.

Using Tinker and our annotated dataset, it is easy to see the effect of various thresholding values on both the accuracy and the number of packets transmitted. We used a simple Threshold operator similar to the one shown in Section 2.2. Our filter suppresses transmission until the current data differs from the one most recently sent by a configurable threshold. We tried thresholds from 1 to 5, each unit being a single increment of the analog-to-digital converter's output (equivalent to 0.1mbar).

We evaluated each threshold using the same metrics we described in the previous section. The results are shown in Figure 3. A threshold of 0 indicates no filtering. As seen in the table, using a threshold of 1 reduces eliminates half the transmissions, but its accuracy impact is minor. A threshold of 2 seems to be the best choice: it reduces transmission by 83%, but accuracy remains high at 98.8%.

3.5 Deployment

The first working prototype of ElevatorNet is depicted in Figure 4. Our seven-floor, three-elevator building was outfitted with ten motes. Five of these motes had altimeters—one in each elevator, one near the top of the elevator shaft, and one near the bottom. Additional motes, without altimeters, were mounted near the elevator shaft on each of the other floors. The altimeter-equipped motes recorded the current pressure every 200msec, and reported the most recent 10 samples every 2 seconds. All motes served as relays for data generated by the altimeter-equipped motes.

We chose a simple flooding algorithm for routing for a number of reasons. First, our goal was to have an elevator display on every floor. This would require broadcast of altimeter data from each elevator and calibration site *to* every floor for display. In addition, the mobility of the elevators meant that data could potentially be generated *from* every floor in the network (that is, the elevator might be near a relay on any of the floors when it generates data). We would therefore need to build an any-to-all broadcast tree. Flooding allows trivial any-to-all broadcast without state maintenance, but is usually inefficient. However, flooding is reasonably efficient given our mostly linear topology. In a topology that is *strictly* linear (i.e., nodes can see only one neighbor on each side), flooding is an optimal broadcast algorithm. Our topology is "almost" linear; packets occasionally leak to more than one floor above or below the transmitter.

In our first prototype, the user interface was a laptop with a mote attached as a network interface. The display software listens for both elevator altimetry updates and calibration data, and computes the elevator's position according to the algorithm described in Section 3.3. A large numeric floor number is shown, along with realtime plots of pressure for those interested. We plan to use only a simple 7-segment LED as the display for the final system. Our estimated cost is about $200 per elevator per floor in the final system, assuming that installation takes one day—a savings of more than 90% over the wired version. (This number does not count the recurring cost of battery replacement.)

There are two major hurdles that remain before a permanent deployment. The first is energy. There is no easy or safe way for the motes in the elevators to receive line power. Our first prototype lasted about 10 days using Tmote's standard pair of alkaline AA batteries. We have studied power saving strategies (Section 3.4), but have not yet gathered enough data to estimate their impact on system lifetime.

The second major hurdle is theft. In the month our prototype was installed, six motes were stolen from elevators. (The privacy of an elevator must create opportunistic thieves: none of the motes placed in lobbies were stolen.) This was surprising, as our building is not open to the public and is primarily staffed with other researchers. We tinkered with different ideas: transparent vs. opaque packaging, obviousness vs. concealment, notes containing descriptions of the experiment and asking people not to steal. The most recent note even offered a free mote to anyone stopping by our lab. The thefts have continued. We consider this to be the most significant obstacle to ElevatorNet's permanent deployment.

4. RELATED WORK

Tinker draws inspiration from a long history of other systems that help the designers of complex systems decompose a problem into at varying levels of abstraction, allowing them to solve problems abstractly before delving into details. This idea is perhaps best exemplified in the mature field of hardware design: VHDL [16], the hardware description language, allows specification first at the behavioral level, then the dataflow level, and finally the structural level. In the area of software, similar design methodologies have been proposed, such as in Lamport's TLA+ [7].

Although Tinker is an imperative language, it is probably most similar to declarative query and data flow languages. In the sensor network space, examples include TAG [12], Cougar [19], and Region Streams [14]. These systems, for the most part, are meant to facilitate fast programming in a well-known set of application spaces, using an set of operators that have efficient distributed im-

plementations. We believe such tools are less well suited for the early stages of exploration and discovery when designers are first trying to understand which application space they're *in*.

As mentioned earlier, Tinker also overlaps with sensor network simulators, including TOSSim [10], EmStar [4], and Avrora [17]. As we argued in Section 1, the feature that these simulators allow simulation of real code can also be a burden: they force all the implementation details to be worked out for every candidate design. Tinker is not a replacement, but a complement to such tools; it allows a designer to rapidly iterate through abstract designs before working through the details of one that looks promising.

Our work's focus is on the design methodology and toolkit more than the end result. Tinker is more general than determining the problem of elevator position. However, past projects have used networked sensors in elevators. Lease and Eddo's SmartElevator [8] similarly addresses the retrofit of a building that has elevators without position displays. Their motivation was similar to ours: when the elevators are under high load, riders grow impatient and call both elevators, exacerbating the delay. SmartElevator used light sensors attached to the elevator position indicators on the first floor lobby. Lester *et al.* [9] describe a system which infers user activity from observations made by small, wearable sensor package. They use a barometric altimeter for classifying activities such as riding up or down an elevator, or using stairs. However, they do not speak to the long-term calibration required to maintain an accurate position estimate over long time periods—a more difficult problem than recognizing instantaneous relative motion.

5. CONCLUSIONS AND FUTURE WORK

We have presented *Tinker*, a high-level design tool that aids the exploration of the design space in sensor network applications. Tinker is targeted at applications that require real-time assignment of semantic meaning to data, rather than just data storage. Tinker lets designers explore the application space before working through the details of the algorithm that seems most promising. Complementary to real-code simulators, which require every detail of an implementation to be specified before giving any intuition of the result, Tinker is a more abstract tool meant for use *before* code-writing begins.

In this paper, we have also described *ElevatorNet*, an application that exemplifies the need for meaningful data interpretation, rather than just data transport and storage. ElevatorNet senses an elevator's position using barometric altimetry; we achieved 99.57% accuracy using an algorithm that was carefully and iteratively refined using Tinker's simulation and visualization features. As expected, the actual implementation required new code to be written, but required no further tuning of the algorithm.

In the future, we would like to extend Tinker, adding features that make it easier to (gradually) understand the effects of various real-world implementation problems: lack of time synchronization, for example. The process of moving our first application from a Tinker simulation to an implementation was instructive in highlighting such areas that Tinker let us ignore for too long. There is a delicate balance between deferring implementation details for later and allowing a crucial detail to be overlooked.

Our future-work aspirations are also to continue to design and build compelling sensor network applications that treat data as data, rather than as a payload. It's all about the data!

6. REFERENCES

[1] Anish Arora, Prabal Dutta, Sandip Bapat, Vinod Kulathumani, and Hongwei Zhang. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks (Elsevier)*, 46(5):605 – 634, 2004.

[2] David E. Culler, Jason Hill, Philip Buonadonna, Robert Szewczyk, and Alec Woo. A network-centric approach to embedded software for tiny devices. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 114–130, London, UK, 2001. Springer-Verlag.

[3] J. Heidemann D. Estrin, R. Govindan and S. Kumar. Next century challenges: scalable coordination in sensor networks. In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking*, pages 263–270, Seattle, WA USA, 1999.

[4] Lewis Girod, Jeremy Elson, Alberto Cerpa, Thanos Stathopoulos, Nithya Ramanathan, and Deborah Estrin. Emstar: a software environment for developing and deploying wireless sensor networks. In *Proceedings of the 2004 USENIX Technical Conference*, Boston, MA, 2004.

[5] Wen Hu, Nirupama Bulusu, Chun Tung Chou, Sanjay Jha, Andrew Taylor, and Van Nghia Tran. A hybrid sensor network for cane-toad monitoring. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 305–305, New York, NY, USA, 2005. ACM Press.

[6] Aman Kansal, Eric Yuen, William J. Kaiser, Gregory J. Pottie, and Mani B. Srivastava. Sensing uncertainty reduction using low complexity actuation. In *IPSN'04: Proceedings of the third international symposium on Information processing in sensor networks*, pages 388–395, New York, NY, USA, 2004. ACM Press.

[7] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[8] Matthew Lease and Guy Eddon. Smartelevator: Revitalizing a legacy device through inexpensive augmentation. In *ICDCS Workshops*, pages 254–259. IEEE Computer Society, 2003.

[9] Jonathan Lester, Tanzeem Choudhury, Nicky Kern, Gaetano Borriello, and Blake Hannaford. A hybrid discriminative-generative approach for modeling human activities. In *Proc. IJCAI-05*, 2005. to appear.

[10] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM Press.

[11] Mark Lutz. *Programming python*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[12] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks, 2002.

[13] Miklos Maroti, Gyula Simon, Akos Ledeczi, and Janos Sztipanovits. Shooter localization in urban terrain. *Computer*, 37(8):60–61, 2004.

[14] Ryan Newton and Matt Welsh. Region streams: functional macroprogramming for sensor networks. In *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks*, pages 78–87, New York, NY, USA, 2004. ACM Press.

[15] Mohammad Rahimi, Rick Baer, Obimdinachi I. Iroezi, Juan C. Garcia, Jay Warrior, Deborah Estrin, and Mani

Srivastava. Cyclops: In situ image sensing and interpretation in wireless sensor networks. In *Proc. IEEE SenSys*, pages 192–204, 2005.

[16] Moe Shahdad. An overview of vhdl language and technology. In *DAC '86: Proceedings of the 23rd ACM/IEEE conference on Design automation*, pages 320–326, Piscataway, NJ, USA, 1986. IEEE Press.

[17] Ben L. Titzer and Jens Palsberg. Nonintrusive precision instrumentation of microcontroller software. In *LCTES'05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 59–68, New York, NY, USA, 2005. ACM Press.

[18] I. Vasilescu, K. Kotay, D. Rus, P. Corke, M. Dunbabin, and P. Schmidd. Data collection, storage and retrueveal with an underwater sensor network. In *Proc. IEEE SenSys*, pages 154–165, 2005.

[19] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, 2002.