On Compressing Data in Wireless Sensor Networks for Energy Efficiency and Real Time Delivery

Tommy Szalapski

Sanjay Madria
Missouri University of Science and Technology, madrias@mst.edu

Distrib Parallel Databases (2013) 31:151–182
DOI 10.1007/s10619-012-7111-5

On compressing data in wireless sensor networks for energy efficiency and real time delivery

Tommy Szalapski · Sanjay Madria

Published online: 21 September 2012

Abstract Wireless sensor networks possess significant limitations in storage, bandwidth, processing, and energy. Additionally, real-time sensor network applications such as monitoring poisonous gas leaks cannot tolerate high latency. While some good data compression algorithms exist specific to sensor networks, in this paper we present TinyPack, a suite of energy-efficient methods with high-compression ratios that reduce latency, storage, and bandwidth usage further in comparison with some other recently proposed algorithms. Our Huffman style compression schemes exploit temporal locality and delta compression to provide better bandwidth utilization important in the wireless sensor network, thus reducing latency for real time sensor-based monitoring applications. Our performance evaluations over many different real data sets using a simulation platform as well as a hardware implementation show comparable compression ratios and energy savings with a significant decrease in latency compared to some other existing approaches. We have also discussed robust error correction and recovery methods to address packet loss and corruption common in sensor network environments.

Keywords Wireless sensor network · Real-time · Compression · Latency

Communicated by Dipanjan Chakraborty.

This research is supported by DOE grant number P200A070359.

T. Szalapski · S. Madria (✉)
Department of Computer Science, Missouri S&T, Rolla, MO 65401, USA
e-mail: madrias@mst.edu

T. Szalapski
e-mail: T.M.Szalapski@mst.edu

1 Introduction

Many real-time systems incorporate wireless sensor networks (WSNs) into their infrastructure. For example, some airplanes and automobiles use wireless sensors to monitor the health of different physical components in the system, security systems use sensors to monitor perimeters and secure areas, security forces use sensors to track troops and targets. It is well known that wireless sensor networks possess significant limitations in processing, storage, bandwidth, and energy. Therefore a need exists for efficient in-network data compression algorithms that do not require delays in processing or communication while still reducing memory and energy requirements.

The idea of data compression has existed since the early days of computers [1–3], many new data compression schemes [4–8] for wireless sensor networks have been proposed recently to address various constraints and limitations in wireless sensor networks. These schemes address specific challenges and opportunities presented by sensor data and provide significant reductions in required storage, bandwidth, and power. However, most of these methods require a fair amount of data to be collected before compressing, which is not suitable for many real-time sensing applications such as those mentioned above.

We propose TinyPack, a suite of data compression protocols for real-time sensor network applications. TinyPack reduces the amount of data flowing through the wireless network, optimizes bandwidth usage, and decreases en without introducing delays. First the data is transformed by expressing the sensed values as the change in value from the previous sensed data. This is referred to as delta compression. We demonstrate its effectiveness for any generic real-time sampled dataset. Second, the individual delta values are then further compressed using a derivative of Huffman coding [1]. Huffman codes express more frequent data values with shorter bit sequences and less frequent values with longer ones. The codes are generated and updated dynamically so no delay occurs. TinyPack is a lossless compression algorithm where the data can be decompressed at the sink or base station without any loss of granularity or accuracy.

Standard Huffman [1] and Adaptive Huffman [2] coding have a high RAM overhead and require transmitting either the entire tree or several copies of a 'new symbol' code, thus making them ineffective in a WSN environment. We begin with a static initial code set similar to the one used in the LEC algorithm [7]. We then examine two different methods of adapting the codes. For datasets where the range of possible values is relatively low compared to the storage capability of the sensors, the actual frequencies can be counted and used to regularly update the codes. For data with a high (or unknown) variance or low RAM environments the frequencies can be approximated using running statistics on the data stream. This method easily scales to be effective on any size data set with any range of possible values. We also use the notion of an all-is-well bit and perform some analysis of error detection constructs.

We compare the results to the performance of the Deflate algorithm (used in gzip and most operating systems) and S-LZW [6] to measure quality of the compression. S-LZW is an adaptation of standard LZW compression specifically designed for sensor networks. S-LZW is a string based compression scheme which defines new characters for common sequences of characters. It is designed to function well for any generic sensor dataset and is very effective at compression and energy reduction. Several variations of S-LZW are developed in [6]. In an effort to be fair we have chosen the variation that performs best for each dataset studied. We also compare with the LEC algorithm [7] which supports real-time data. Experiment and simulation results show a significant reduction in bandwidth, latency, and energy consumption compared to the other methods. One of the proposed algorithms also reduces RAM and processor usage while the others show a further reduction in bandwidth, energy, and latency at the cost of increasing the memory and processing requirements.

In summary, this paper makes the following contributions:

- An improved set of static codes optimized for sensor data and computational efficiency in processing.
- Algorithms for hybrid adaptations of delta and Huffman compression which significantly reduce latency and RAM requirements over traditional Huffman codes while achieving comparable and improved compression ratios and energy efficiency compared to other existing methods.
- An additional use of an all-is-well bit that further increases compression performance and efficiency.
- A novel and effective error detection and recovery method to handle missing and corrupted packets.
- Extensive experiments comparing several performance metrics considering various approaches using many different real sensor data sets using simulation as well as a hardware platform.

2 Background

2.1 Huffman trees

Huffman-style coding [1] converts each possible value into a variable length string (sequences of bits) based on the frequency of the data. Higher frequency values are assigned shorter strings. The more concentrated the data is over a small set of values, the more the data can be compressed. Huffman codes can be generated by building a binary tree where the nodes at each level are ideally half as frequent as the nodes at the next level up. For example, the values and frequencies in Table 1 generate the codes using the Huffman tree in Fig. 1. Huffman codes were shown to be optimal for symbol by symbol compression in [1].

2.2 Temporal locality and delta values

Real-time wireless sensor networks generally exhibit temporal locality (data from readings taken in a small time window are correlated). Any type of data which changes in a continuous fashion will be temporally located such as temperature, location, voltage, velocity, timestamps, etc. In fact, it can be demonstrated that any sensor sensing at non-random intervals will either generate temporally located data or random noise.

Table 1 Huffman codes

Value	Frequency	Code
−7	14653	111111
−6	16661	111101
−5	19983	111011
−4	23760	111001
−3	31124	11011
−2	35636	11001
−1	88845	101
+0	350429	0
+1	87956	100
+2	38942	11000
+3	31809	11010
+4	20563	111000
+5	17241	111010
+6	14171	111100
+7	12716	111110

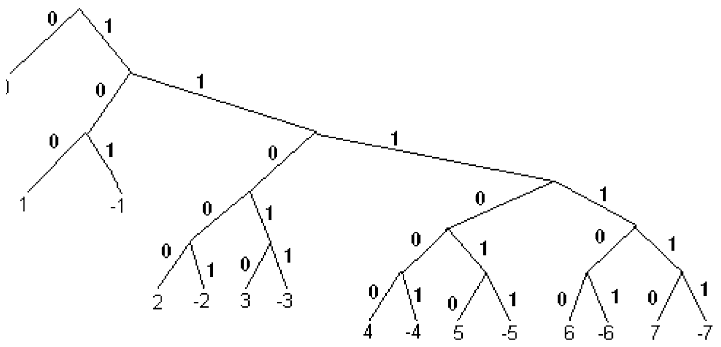


Fig. 1 Huffman tree

Consider an arbitrary sensor sensing a stream of values $\{v_1, v_2, \ldots, v_{2N}\}$ sensed at times $\{t_1, t_2, \ldots, t_{2N}\}$ where N is an integer. Assume that the values are not correlated. Then sampling at $\{t_1, t_3, \ldots, t_{2N-1}\}$ and $\{t_2, t_4, \ldots, t_{2N}\}$ would yield completely different values. Thus, offsetting the sample period would generate entirely different data. Therefore, excluding applications which generate pure noise, we can assume that successive readings at each sensor will be correlated. Delta compression (storing the data as the change in value from the previous reading) would then increase the frequency of certain values thus increasing the compressibility of the data.

Note that this does not apply to event driven sampling (where time between samples is random) such as a sensor that measures the speed once for each passing automobile. These applications do not necessarily exhibit temporal locality and were not included in this study.

Table 2 S-LZW with mini-cache

Encoded string	New output	New dict. entry	Mini-cache changes	Total bits: LZW	Total bits: mini-cache
A	0,65	256-AA	0-256, 1-65	9	10
AA	1,0	257-AAA	1-257	18	15
A	0,65	258-AB	1-65,2-258	27	25
B	0,66	259-BA	2-66,3-259	36	35
AAA	0,257	260-AAAB	1-257,4-260	45	45
B	1,2	261-BC	5-261	54	50
C	0,67	262-CC	3-67,6-262	63	60
C	1,3			72	65

2.3 Frames

In delta compression (as with most compression schemes), a dropped packet can render following packets useless or at least complicated to decompress. Thus in systems where data loss is probable, data should be compressed and sent in chunks (usually called frames). Additionally, in sensor networks, data characteristics can change drastically as time progresses. Therefore, sending independently compressed frames of data also allows additional flexibility for the compression to be more specific to the current state of the system.

3 Related work

3.1 S-LZW

In [6] an adaptation of standard LZW compression is used to address the specific characteristics of a sensor network. S-LZW compresses the data by finding common substrings and using fewer bits to represent them. S-LZW maintains two sets of up to 256 eight-bit symbols: The original ASCII characters and the set of common strings. A bit is appended to the beginning of each encoded symbol to indicate which set it is from. A dictionary is maintained that tracks which string is represented by which eight-bit sequence.

They also propose Sensor-LZW with the notion of a mini-cache to capitalize on the frequent recurrences of similar values in a short time in sensor data. Recent strings are stored with N bits in the mini-cache dictionary where $N < 8$ (for a maximum size of 2^N entries in the mini-cache). An additional bit is appended to the beginning of each symbol to note whether the symbol is from the main dictionary or the mini-cache. Different data sets had different optimal values for N. The cache is implemented as a hash table for efficient lookup times.

Table 2 shows S-LZW and LZW compressing the string AAAABAAABCC using the mini-cache. Since every single character is pre-loaded into the dictionary, the algorithm begins by looking at the first string of two characters in the stream. If the string is in the dictionary, the next character is appended until the string no longer

Table 3 Lec codes

Level	Bits	Prefix	Suffix range	Values
0	2	00		0
1	4	010	0...1	-1.1
2	5	011	00...11	$-3, -2, 2, 3$
3	6	100	000...111	$-7, \ldots, -4, 4, \ldots, 7$
4	7	101	0000...1111	$-15, \ldots, -8, 8, \ldots, 15$
5	8	110	00000...11111	$-31, \ldots, -16, 16, \ldots, 31$
6	10	1110	000000...111111	$-62, \ldots, -32, 32, \ldots, 63$
7	12	11110	0000000...1111111	$-127, \ldots, -64, 64, \ldots, 127$

has a dictionary entry. Then that new string is added to the dictionary and the known string (the new string minus the last character) is encoded into the output. The new output column shows a 1 and the mini-cache location if that symbol was in the cache or a 0 and the dictionary location otherwise. The other columns show the new entries in the dictionary and mini-cache and the total number of bits required for compression without or with the cache. Note that without the cache every symbol is exactly nine bits.

For example, for the first line of Table 2 the compressor begins by looking at the first character of the string "A." Since "A" is a single character it is already in the dictionary and the compressor looks at the string "AA." That string is not in the dictionary so it is added to the end (location 257) and the single character "AA" is encoded (as the integer 65) and the algorithm continues with the second "AA" as the next character in the stream. Since "AA" was not in the mini-cache the output comes from the dictionary and both "AA" and "AA" are added to the cache.

3.2 LEC

A lightweight sensor network compression technique, LEC, is presented in [7]. LEC compresses a stream of integers by encoding the delta values with a static, predetermined set of Huffman codes. For the values in a stream, the initial value is encoded as its difference from 0 and each successive value is encoded as its difference from the previous value. The codes are constructed by concatenating prefix and a suffix bits to represent the change value. Fewer bits are used for the smaller changes under the assumption that values typically change relatively slowly over time. The static codes are shown in Table 3 with anything past level 7 following the pattern of the last three levels.

For example, a 0 value would be encoded as "00" ("00" prefix and no suffix) and -3 would be encoded as "01100" ("011 prefix and "00" suffix).

If it is known that the change values will not fall outside of a certain range, then the '0' bit in the prefix for the last level can be removed. For example in Table 3 the prefix for level 7 could be "1111" if -127 and 127 were the minimum and maximum possible change values.

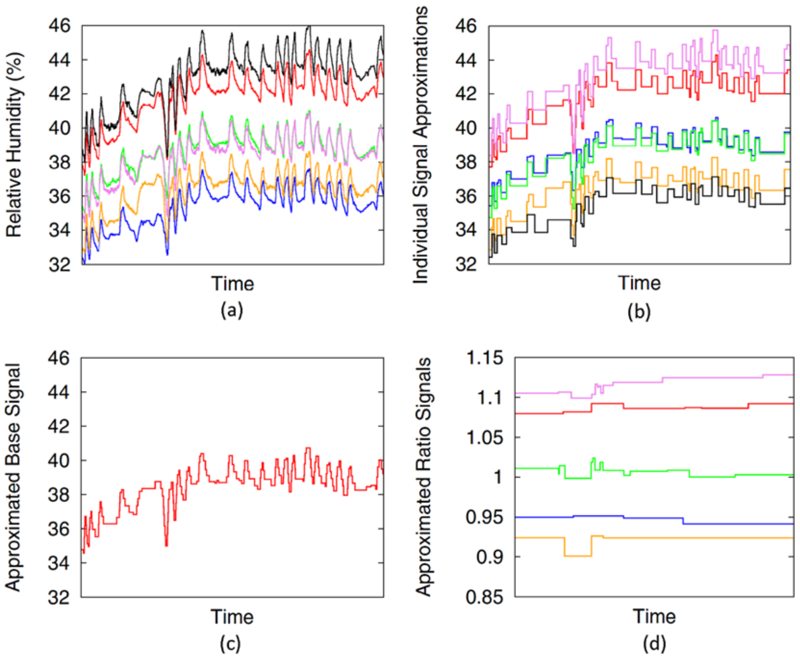


Fig. 2 GAMPS example

3.3 GAMPS

Many lossy compression schemes have also been proposed such as [8]. GAMPS compresses the data from multiple sensors by grouping sensors with correlated values. The signals are approximated keeping within a parameterized maximum error. The Facility Location problem is then used to groups the sensors with the highest correlations and select baseline sensors which best represent the group. The values from the remaining sensors in each group are expressed as a ratio of the value of the baseline.

An example is shown in Fig. 2. Graph (a) shows relative humidity signals from different sensors. In graph (b) the signals have been approximated. Graph (c) shows the fourth signal from graph (b) selected as the baseline for the group. The final graph (d) shows each of the other five signals as a ratio of the baseline signal. The data in graphs (c) and (d) is then identical to the data in (a) within some error threshold but can be compressed much more than the original data.

GAMPS achieves excellent compression ratios with low maximum error but requires that all the data be collected before compression and so is not suited for applications which require no loss or for the compression to be performed in real time.

3.4 Pipelined in-network processing

Other schemes have been introduced which depend on the network topology and routing. In [4] compression is achieved using pipelining. Data is gathered at each aggregation node in a buffer for some amount of time. During that time, several data

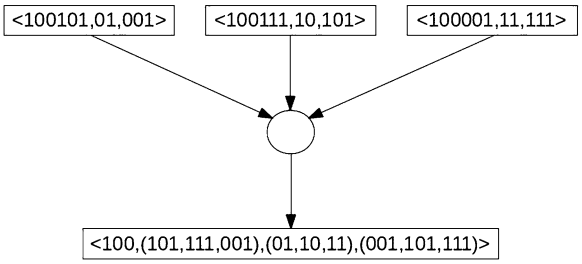


Fig. 3 Pipelined compression

Table 4 Value indicated by order

Packet permutation	Integer value
N1,N2,N3	0
N1,N3,N2	1
N2,N1,N3	2
N2,N3,N1	3
N3,N1,N2	4
N3,N2,N1	5

packets with a matching prefix are combined into one. Following the prefix in the packet is a suffix list which gives the unique suffix to the common prefix from each of the original packets. This scheme is illustrated in Fig. 3. Three packets each containing three items of data are compressed on the first item with a prefix of length three, the other two items remain uncompressed. This reduces the data size from 33 bits to 27 bits.

The size of the prefix is determined by the user of the application and remains static. The shared prefix system can also be used for timestamps and sensor IDs to maximize the reductions in size.

This scheme can be very effective if there is much redundancy inherent in the value prefixes; however, the compression is only done at aggregating nodes and depends on sample rates to be very effective.

3.5 Coding by ordering

Another routing method is proposed in [5] where the order of packets collected at an aggregation node can indicate the value sensed at a different node. A packet containing the data tuples from n sensors can be arranged in a total of $n!$ unique permutations. If the number of possible sensed values is relatively small, these permutations can be used to recreate dropped values from one or more sensors (see Table 4).

If there are n sensor nodes in a network and a packet at an aggregation is sent values from m different nodes, assume that out of those m nodes a total of l nodes' values are dropped and encoded. Given only the $(m-l)$ values, there are $(n-m+l$ choose $l)$ possible combinations of IDs the dropped nodes can have. If there are k possible data values, there are k^l possible combinations of values and IDs. Since there are $(m-l)!$ possible permutations within the packet, l can be chosen as large as is possible without violating the following inequality

$$(m-l)! \geq (n-m+l \; choose \; l)k^l$$

For example, when $n = 256$, $k = 16$, and $m = 100$; l could be set as high as 44, so only 56 % of the data would need to be sent. This scheme, however, performs well only when n is relatively large compared to k. If there is a wide range of possible data values, then some form of tolerated error would need to be introduced to accomplish any amount of reduction.

3.6 Summary

We compare all the previously listed algorithms and the algorithm presented in this paper (TinyPack) across a number of compression algorithm characteristics in Table 5.

The algorithms presented in this paper and used for comparison concern lossless compression which can be achieved in real time at the sensing node.

4 Experimental data sets used

The data sets used for simulation were pulled from a wide variety of domains which utilize wireless sensor networks including environment monitoring, tracking, structural health monitoring, and signal triangulation. All except the environment monitoring data are from applications where low latency is critical. All are from real deployments of wireless sensors for academic, military, and commercial purposes. In every experiment, the entire datasets were used.

Environment monitoring data was drawn from the Great Duck Island [9] and Intel Research Laboratory [10] experiments. On the island 32 sensors monitored the conditions inside and outside the burrows of storm petrels measuring temperature, humidity, barometric pressure, and mid-range infrared light. The Intel group deployed 54 sensors to monitor humidity, temperature, and light in the lab. Approximately 9 million sensed values were generated on the island and over 13 million from the lab.

For tracking, data was taken from two different studies. Princeton researchers in the ZebraNet project [11] tracked Kenyan zebras generating over 62,000 sensor readings. The U.S. Air Force's N-CET [12] project tracked humans and vehicles moving through an area.

The structural health data is comprised of nearly half a million packets send by a network of 8 sensors fused to an airplane wing in a University of Colorado study [13]. Half the data was generated by a healthy wing and the other half by a wing with simulated cracking and corrosion.

Signal triangulation data came from another portion of the N-CET project, in which a network of sensors mounted on unmanned aerial vehicles intercepted and collaboratively located the sources of RF signals.

Table 5 Characteristics of sensor compression techniques

Characteristic	S-LZW	LEC	GAMPS	Pipelined	Coding by ordering	TinyPack
Runs on a single sensor	Yes	Yes	No	No	No	Yes
Relies on temporal locality	Sometimes	Yes	Yes	No	No	Yes
Relies on spatial locality	No	No	Yes	Yes	No	No
Collect data prior to compressing	Some	None	All	Some	None	None
Algorithm adapts as data changes	Yes	No	Yes	No	No	Yes
Requires time synchronization	No	No	Yes	No	Yes	No
Requires related sampling intervals	None	None	None	Similar	Identical	None
Achieves lossless compression	Yes	Yes	No	Yes	Yes	Yes
Loss due to dropped packets or errors	Entire Frame	Entire Frame	Single packet	Single packet	One per sensor	Entire Frame
Incorporates error detection	No	No	No	No	No	Yes

Table 6 Initial default codes

Value	+0	−1	+1	−2	+2	−3	+3
Code	1	011	010	00101	00100	00111	00110

5 Our proposed approaches

We propose multiple versions of our TinyPack compression algorithm. First we introduce a static set of initial codes which are used as a starting point for the other compression methods. These codes by themselves provide good compression with excellent efficiency. Next we achieve greater compression at the cost of some RAM and processing by maintaining dynamic frequencies of the streamed values. The third approach approximates the frequencies with running statistics on the data, significantly decreasing the RAM requirements while only slightly increasing the size and processor utilization. We modify each of the above approaches by adding an all-is-well bit that gives a small boost to the compression ratio. We conclude by discussing error detection, how to adjust for real numbers instead of integers, and experimental results.

5.1 TinyPack initial frame static codes (TP-Init)

We begin with a set of initial codes similar to those used in LEC; however, the static codes used in LEC were optimized for JPEG compression whereas the TinyPack initial codes are designed to perform well on time-sampled sensor data with absolute minimum processing time required.

Since we are using delta compression, the data is expressed as the change in value from the previous sample. The reported values can be positive or negative. In many applications such as temperature sensing the values are cyclic so the frequency of positive changes is similar to the frequency of negative changes. In general, highest frequencies appear in the smaller values (e.g. temperature usually changes fairly slowly causing most changes reported to be small). Also the set needs to scale to any number of values. Based on these characteristics, we construct an initial set of codes (see Table 6).

With all other values continuing the pattern: Define B as the base of the delta value d where

$$B = \begin{cases} \mathit{floor}(\log_2(|d|) & |d| > 0 \\ -1 & d = 0 \end{cases}$$

The code C is constructed as a string of $2B + 3$ bits. The first $B + 1$ bits are 0s followed by the binary representation of $|d|$ (which will be $B + 1$ bits), and a sign bit. For example, if d is 57 then B is 5. Then C is constructed as six 0 bits, followed by the binary representation of $|57|$ (i.e. 111001), followed a 0 sign bit since 57 is positive. The entire code C is then 0000001110010.

If the minimum and maximum allowed for the value are known, then the 1 bit in the center can be removed for the longest set of codes. For example, in the codes for −3 to +3 above, if the 1 bit in the center of the codes for −2, +2, −3, and +3 was removed, the leading 00 would be enough for the decoder to accurately decode those

Table 7 Default codes

Level	Bits	Prefix	Suffix range	Values
0	1	1		0
1	3	01	0…1	−1.1
2	5	001	00…11	−3, −2, 2, 3
3	7	0001	000…111	−7, …, −4, 4, …, 7
4	9	00001	0000…1111	−15, …, −8, 8, …, 15
5	11	000001	00000…11111	−31, …, −16, 16, …, 31
6	13	0000001	000000…111111	−62, …, −32, 32, …, 63
7	14	0000000	0000000…1111111	−127, …, −64, 64, …, 127

Algorithm 1 FloorLog2Byte(d)

```
Objective: Calculate the base of a value
Input: Delta value $d$
Output: The base $B$ of value $d$
  $B = 0$
  If $d = 0$
    $B = -1$
  Else
    $d := |d|$
    If $d >=$ b10000
      rightBitShift($d$, 4)
      $B := B$ bitwiseOr b100
    End If
    If $d >=$ b100
      rightBitShift($d$, 2)
      $B := B$ bitwiseOr b10
    End If
    If $d >=$ b10
      $B := B$ bitwiseOr 1
    End If
  End If
```

symbols. The initial static codes for values ranging from −127 to 127 are shown in Table 7. The leading 1 bit in the number is considered to be part of the prefix since it is static for the entire level of the tree.

Using bitwise operators the floor (round down) of log base 2 can be calculated in logarithmic time with respect to the maximum value of d using Algorithm 1. The example shows getting the base for a one byte value. The notation bxxxx is used to indicate a binary number, for example b10000 = 16.

The value is then bit shifted to fill in the $B + 1$ prefix bits and appended to the output stream.

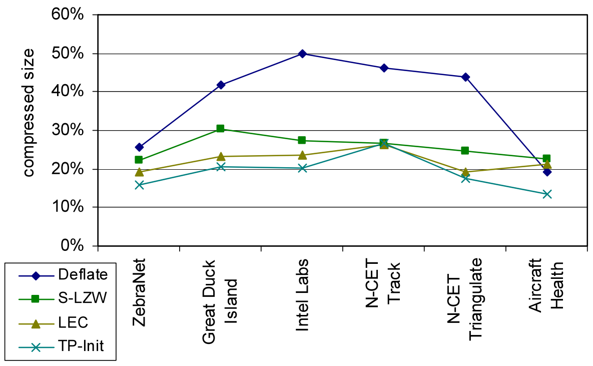


Fig. 4 Initial codes compared to Deflate, S-LZW, and LEC

In order to test the validity of this initial default set, we compressed each of the datasets using only these codes. Figure 4 shows the results of the TinyPack initial codes (TP-Init) compared to the standard Deflate algorithm, S-LZW, and the LEC codes.

For all the datasets our initial codes actually compressed slightly better than any of the other methods except for the N-CET Track dataset where S-LZW, LEC, and our initial codes had nearly identical performance. This is due to the high degree of variance in that dataset. As expected, the Deflate algorithm, which does not specifically target sensor network data, performed significantly worse for most of the datasets. The ZebraNet and aircraft health datasets both contain significant runs of unchanging data which the Deflate algorithm takes advantage of so it performed relatively well on those datasets compared to the sensor network specific algorithms.

5.2 TinyPack with dynamic frequencies (TP-DF)

In order to use Huffman-style compression, the frequencies of the different data values must be known. However, in real-time systems there is often no time to collect all the data and count the total frequencies of all the values before sending the currently collected data. To avoid the need to transmit them, the frequencies from the last frame of data can be used. The frequencies are calculated both at the source and the destination to avoid the need to transmit the frequency tables. The trees and codes are updated at the beginning of each frame. Naturally, values that are in the possible range but do not appear in a frame are assigned a frequency of zero.

Since the values are typically densely clustered around 0 and sparsely scattered far from 0, the frequencies are stored in a hash table. The hash for the value is the last eight bits using 2's compliment for negative numbers so the values from -128 to 127 fit neatly into the table. The hash table is chained and colliding values are stored in a list in the hash table bucket. This keeps the RAM requirements reasonably low while still allowing for fast lookups.

In order to capitalize on the dynamic characteristics of sensor data we add weight to the most recent values in order that recent occurrences have a higher impact than past occurrences but the history is not entirely forgotten. We replace the frequency table with a weighted frequency table and define a weighting factor M such the occurrence of a new value is given twice the weight of the value observed M samples ago.

Algorithm 2 CountAndEncode(d, n, M, S, F)

```
Objective: Maintain count of frequencies and encode data
Input: Delta value $d$, count $n$, weighting factor $M$, frame size $S$, frequency table $F$
Output: Frequency table updated and code appended to stream
If Hash($d$) in $F$
    $F[d] := F[d] + 2\^{}(n/M)$
Else
    $F[d] := 2\^{}(n/M)$
End If
$C :=$ LookupCode($d$)
AppendToStream($C$)
$n := n + 1$
If $n = S$//New frame
    $n = 0$
    For every $F[x]$ in $F$
        $F[x] := F[x]/(2\^{}(S/M))$
        If $F[x] < .001$
            $F[x] := 0$
        End If
    End For
    UpdateCodes($F$)
End If
```

The weighted frequency $F[d]$ for a value d appearing in the nth sample is updated by the following equation:

$$F[d] = F[d] + 2^{\frac{n}{M}}$$

In our experiments we set M equal to the one quarter of the frame size. At the end of a frame when the tree is updated, the weighted frequencies are normalized to reset n to 0 and prevent overflow. Also any values with a normalized frequency less than .001 are assigned a frequency of 0 and removed from the list of counted values. Algorithm 2 runs for each delta value in a sensed vector.

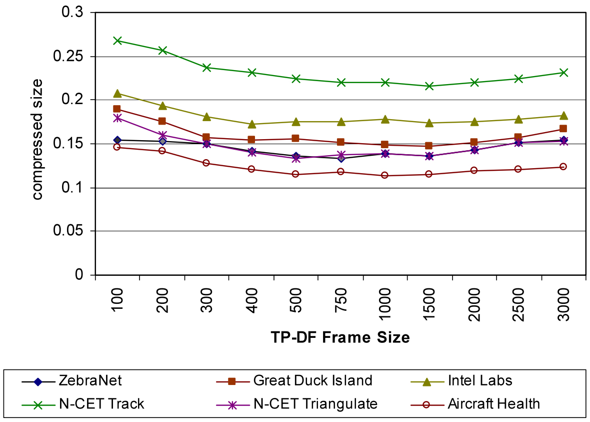
We ran TP-DF on all the datasets with a varying frame size. Results are shown in Fig. 5. When the frame size was small, the overhead for creating a new frame had a significant impact on the compressed size. When the frame size was very large, the codes were not updated frequently enough to keep up with the dynamic characteristics of the data, thus again negatively impacting the compression size.

Frame sizes between 500 and 1500 samples per sensor had roughly the same impact. Thus, for our experiments, we set the frame size to be 512 samples.

5.3 TinyPack with running statistics (TP-RS)

In cases where the number of possible values is very high or memory is very limited, storing the frequency table can be too costly since a standard Huffman tree on that much data would require more RAM than many sensors have available. For example, storing the frequency table for a single 4-byte integer if the values covered the entire

Fig. 5 Frame size analysis for TinyPack with dynamic frequencies



possible range would require over 8 MB of RAM while Crossbow Technology's [14] popular Mica2 and MicaZ motes have less than 1 MB of total memory. In these cases the frequencies can be approximated by maintaining running statistics such as the mean and standard deviation. Because we use delta values, it is not necessary to know the distribution of the data; only the distribution of how the data changes. This remains much more consistent in all of our datasets.

Beginning with the average and standard deviation that the default codes would produce the running average and standard deviation can be calculated over a window of size W. The running average $E(d)$ updates when the nth value d is sampled by the simple equation:

$$E(d)_n = \frac{1}{W}d_n + \frac{W-1}{W}E(d)_{n-1}$$

In the same way, the average of the squares of the values can be maintained. We can compute the standard deviation σ using the well known formula:

$$\sigma = \sqrt{E\left(d^2\right) - \left(E(d)\right)^2}$$

The frequency of a value occurring in a stream divided by the total number of values in the stream is referred to as the probability of that value. In a Huffman tree the probability of each leaf node is the probability of that value occurring in the stream and the probability of a non-leaf node is the sum of the probabilities of each child node. The probability of the root is 1. The probability of each node was shown by Shannon [15] to be ideally half the probability of its parent, so the level of a node in the tree should be $-\log_2(P)$ where P is the probability of the node. Using the statistics calculated the probabilities of each value can be approximated. Then the tree can simply be expressed as a table containing the number of leaf nodes that should be at each level. Therefore, the Huffman tree in Fig. 1 can be compressed into Table 8 where the table is stored on the sensor as an array 1-indexed on the tree level.

The code strings for the values can then be generated in logarithmic time.

These codes are generated by creating a base code similar to a prefix for each level in the tree and using the position of each node at its level. The binary base for all

Table 8 Compressed tree

Level	Count
1	1
2	0
3	2
4	0
5	4
6	8

Table 9 Base generation

Level	Count	Binary	Generation	Base
1	1	1	0	0
2	0	0	(0+1)*10	10
3	1	1	(10+0)*10	100
4	3	11	(100+1)*10	1010
5	4	100	(1010+11)*10	11010
6	4	100	(11010+100)*10	111100

nodes at a level in the tree is generated by adding the base and count of the previous level and multiplying by 2 (appending a 0) with the base for the root initialized to 0. For example, suppose the statistics approximated a tree with one node at level 1 and 1, 3, 4, and 4 nodes at levels 3, 4, 5, and 6 respectively for values of 0 to 12. The base generation for these values is shown in Table 9.

The code for a value is generated by adding the value's position in the level to the group's base. Again, all the arithmetic is done in binary. Continuing the above example, the generation for the codes of these values is shown in Table 10.

The probability of a level is computed as the sum of the probabilities of the nodes at that level. Since the probability of a node at level L is ideally 2^{-L}, the probability of a level is defined by:

$$P(L) = \big(Count(L)\big)\big(2^{-L}\big)$$

The probability of the table $P(T)$ is defined as the sum of the probabilities of all the levels. For the table to generate accurate codes, $P(T)$ must be less than one; however, the higher it is, the more compact the code are. Thus, the following relationship should hold (where H is the height of the tree):

$$P(T) = \sum_{L=1}^{H} \big(Count(L)\big)\big(2^{-L}\big) = 1$$

Events such as changes in values are often assumed to follow exponential distributions. Experiments confirmed this in our datasets allowing confidence intervals to be used to approximate the ideal number of nodes at each depth of the tree. The values are assigned to their ideal levels rounding down so that $P(T)$ remains less than 1.

Table 10 Code generation

Value	Level	Position	Base	Generation	Code
0	1	0	0	0+0	0
1	3	0	100	100+0	100
2	4	0	1010	1010+0	1010
3	4	1	1010	1010+1	1011
4	4	2	1010	1010+10	1100
5	5	0	11010	11010+0	11010
6	5	1	11010	11010+1	11011
7	5	2	11010	11010+10	11100
8	5	3	11010	11010+11	11101
9	6	0	111100	111100+0	111100
10	6	1	111100	111100+1	111101
11	6	2	111100	111100+10	111110
12	6	3	111100	111100+11	111111

Algorithm 3 FilterUp(T, H)

```
Objective: Produce optimal codes by getting $P(T) = 1$
Input: Table $T$ where $T$ is simply the array of the counts, Height of tree $H$
Output: $T$ adjusted so that $P(T) = 1$
  $P(T) := 0$
  For $L$ From 1 to $H$
    $P(T) := P(T) + T[L] * 2^\wedge(-L)$
  End For
  For $L$ From 1 to $H - 1$
    //Get the highest number that can possibly move
    move_count := Floor($(1 - P(T))/(2^\wedge(-L - 1))$)
    //Don't move more than are there
    move_count := Max(move_count, $T[L]$)
    //If move_count is 0 the next two lines do nothing
    $T[L] := T[L] +$ move_count
    $T[L + 1] := T[L + 1] -$ move_count
  End For
```

Then the table is adjusted from the top down using Algorithm 3 so that nodes are pushed upward in the tree until $P(T) = 1$.

The window size analysis for the running statistics was almost identical to the frame size results using dynamic frequencies (shown in Fig. 5). Again the experiments were run with a window size of 512.

Figure 6 shows the results of running both the dynamic frequencies (TP-DF) and running statistics (TP-RS) over the datasets compared to the other methods. The running statistics generally performed slightly poorer than dynamic frequencies except

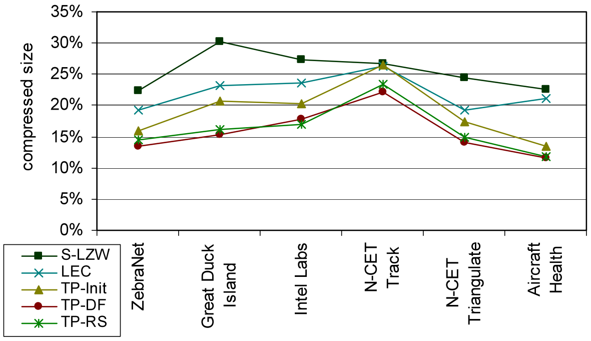


Fig. 6 TinyPack with dynamic frequencies and running statistics

on the Intel Labs dataset. The data in this set is more precise and follows a cleaner statistical pattern than the others.

5.4 All-is-well bit

Most sensor applications send a vector of values (e.g., timestamp, temperature, humidity) at each sampling interval. Often in the data sets studied all the values in a sample were exactly equal to the previous corresponding value. A bit can be appended to the beginning of the packet indicating whether or not this has occurred (obviously if it has, no more data needs to be sent for that packet). In protocols with variable sized packets or packets that are small compared to the size of a vector of readings, this could introduce additional savings. This idea has been used several times previously in sensor networks [16–18].

The datasets were affected differently by adding this. Figure 7 shows the effects of the all-is-well bit (AIW). TP-DF and TP-RS were very similar, so TP-RS was removed to avoid cluttering the graph. In each of the TinyPack algorithms the all-is-well bit improved performance for all the datasets except the aircraft health and N-CET tracking sets. This is due to the higher level of precision in those datasets. The datasets had a very small number of packets where all the values were identical to the previous packet. In general, if the application is designed such that sensed values will rarely be exactly equal to the previous value (as in high precision data), the all-is-well bit should not be used.

Additionally, if the sensors send on a predetermined schedule or if the packet headers contain consecutive sequence numbers, simply refraining from sending data could be used to indicate the same thing as the all-is-well bit. This would remove the overhead so no decision would need to be made whether or not to use it. These intentionally unsent packets would be easily differentiated from actual drops based on the sequence numbers or the error detection discussed in the next section.

5.5 Baseline frequency

In some applications, packets that are uninteresting can be dropped and drops can also occur accidentally. Since the compression of the packets depends on the previous packet, any loss of a packet causes errors that propagate to all the following packets.

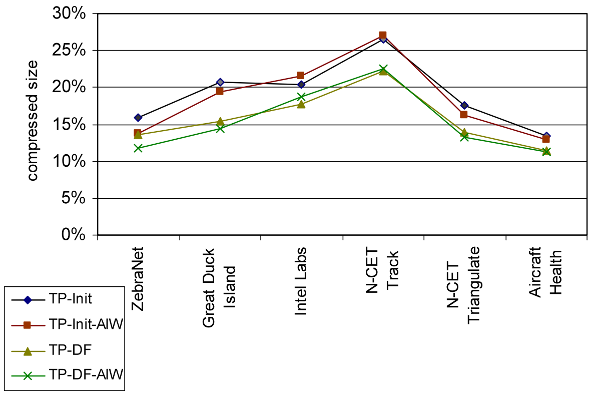


Fig. 7 Effects of all-is-well bit

Instead of reporting the value at each packet as the change in value from the previous packet, we examined the cost of only occasionally changing the baseline of which the change is reported. So instead of every packet being a baseline, baseline packets can be sent at different intervals and all subsequent packets are expressed as the change in value from the last baseline. These baseline packets can then be flagged as high priority so that the application will not drop them. Also in lossy environments, these baseline packets can require acknowledgement to ensure delivery. We experimented with static baseline intervals and using statistics of the data to determine when to send the new baseline. Figure 8 shows the effects on compression of changing the baseline frequency using static intervals and sending a new baseline when the packet size increased above a threshold compared to the average and standard deviation of the previous packet sizes.

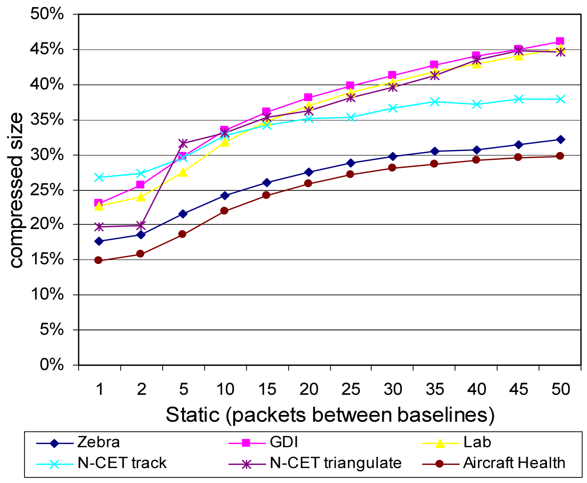
The results for the statistical approach were scaled using the total number of baseline packets sent to calculate the frequency and compared to the results for static frequencies for each of the datasets. The average results were almost identical making the static methods preferable since they require less processing, are more intuitive to implement and parameterize, and were more consistent in their effects.

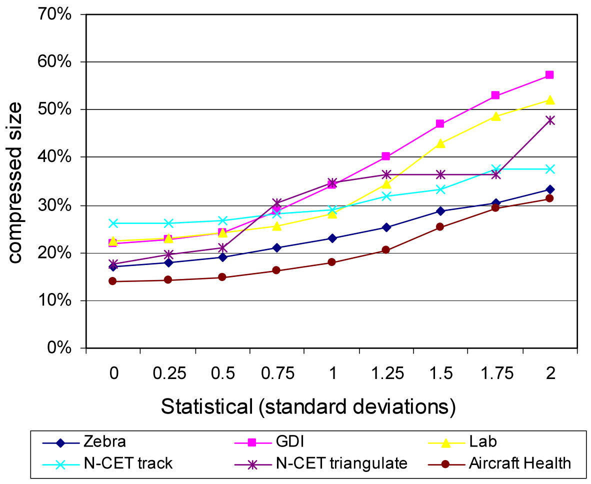
As with most compression algorithms, the data is highly susceptible to dropped or corrupted packets. If one of the baseline packets is dropped or corrupted, then the data following that point would be unable to be decompressed. We experimented on and analyzed the cost of retransmitting baseline packets in scenarios with varying degrees of error. Error detection and correction are discussed in more detail Sect. 7.

Figure 9 shows the cost of retransmission of the dropped baseline packets. As expected, the cost of retransmission drops quickly as the number of packets between baselines increases. The probability of a dropped packet being a baseline and thus requiring retransmission is inversely proportional to the number of packets between baselines resulting in the hyperbolic shape of the cost curve.

Cost of retransmission was directly proportional to error percentage. The graphs for the other error amounts were omitted since the shape of the curves is identical. Figure 10 shows the total size of the transmitted compressed data including retransmissions of dropped baseline packets. This includes dropped retransmissions. For example, with 10 % error, each baseline packet would be sent an average of 1.111 times and with 50 % error, each baseline would be sent an average of twice. As the

Fig. 8 Baseline frequency



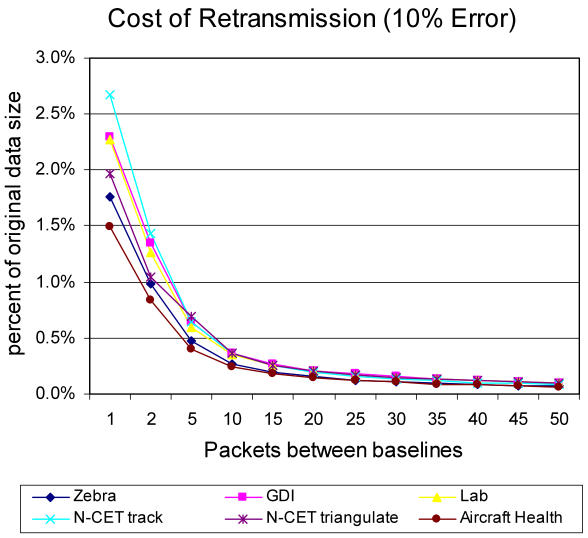


error rate increases, the cost of retransmission increases. As in Fig. 9 the increased cost is greatest when the number of packets between baselines is low. As the number of packets between baselines increases, the added cost becomes negligible and the graphs become identical.

5.6 Working with real values

TinyPack works most effectively with integers. Our approach could fairly intuitively be extended into the real numbers; however, for simplicity in our experiments, we expressed reals as integers. In the case where the real values were rounded in the dataset to some low number of decimal places, we simply shifted the decimal point. In the case of higher precision reals, we split the values into the exponent and mantissa and compressed them separately.

Fig. 9 Retransmission



6 Physical implementation using sensor network test-bed

We implemented the algorithms on a network of seven Mica2 sensors running the TinyOS operating system. One sensor served as the base station for the network and the other sensors were loaded with data from the datasets. The sensors then compressed and sent that data to the base station using each of the different algorithms. All the sensors were time synchronized and sent data using time division multiplexing. For datasets with more than six sensing nodes, experiments were done on the data from six at a time until the data from all sensing nodes had been passed through the network.

Each experiment was run separately in order that the measurement of one metric would not affect the others. For example, if the sensors tracked RAM usage while processor utilization was being measured, the results would be slightly inflated.

6.1 Compression

The results from all the previous compression experiments are combined in Fig. 11 which shows the compressed size of each dataset. Shown are the standard Deflate algorithm used in most operating systems, S-LZW, LEC, and our approaches: The static initial codes (TP-Init), dynamic frequencies (TP-DF), running statistics (TP-RS), and each of the TinyPack methods with the all-is-well bit added (-AIW). As expected TP-DF performed the best in terms of compression compared to the other algorithms. The all-is-well bit increased the performance over some of the datasets.

To summarize, we calculate the entire compression of all the data across every dataset and normalized the results to give equal weight to each dataset in Fig. 12. The all-is-well bit added a slight benefit in the average case although its usefulness depends heavily on the characteristics of the data sensed. As it can be observed, the TinyPack algorithms provide compressed sizes of 11 % to 27 % outperforming the other methods which range from 19 % to 50 %.

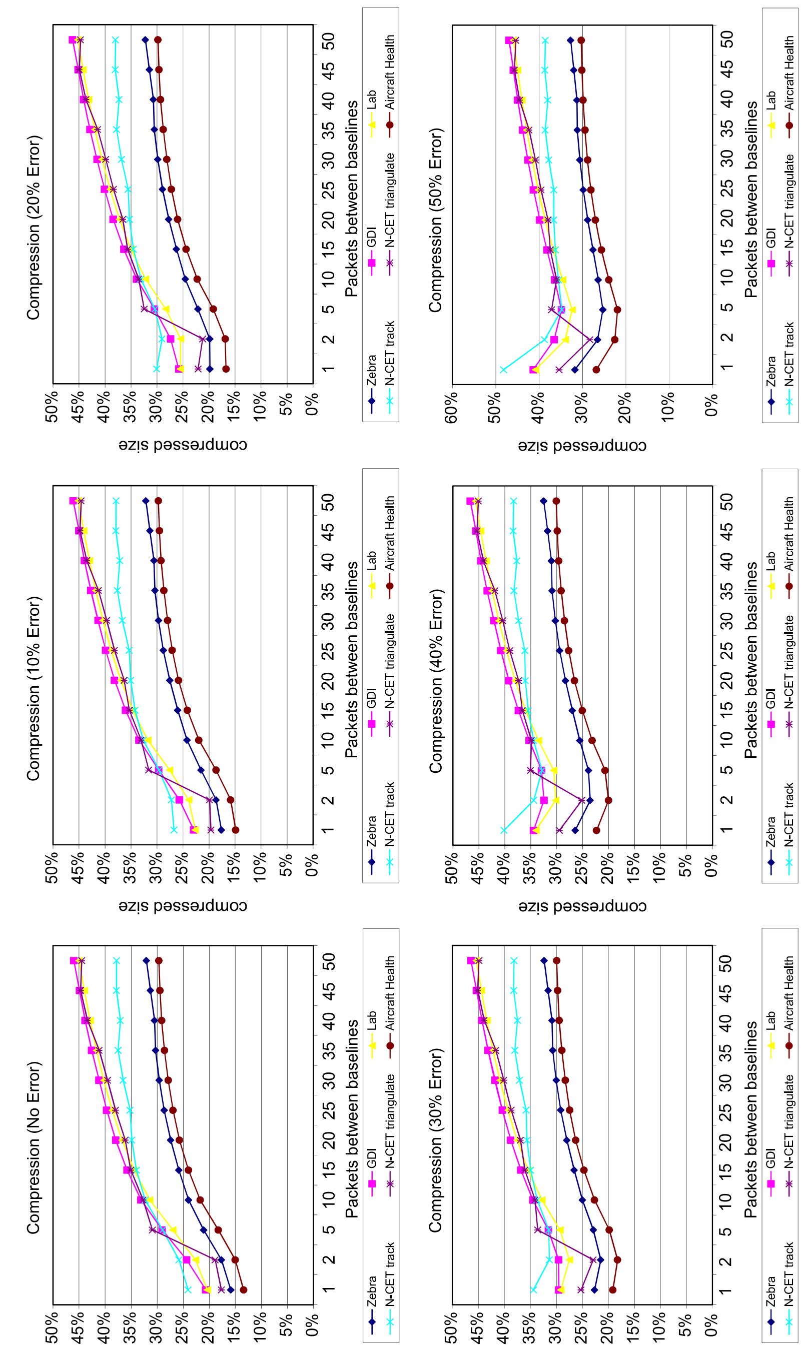


Fig. 10 Compression with retransmission

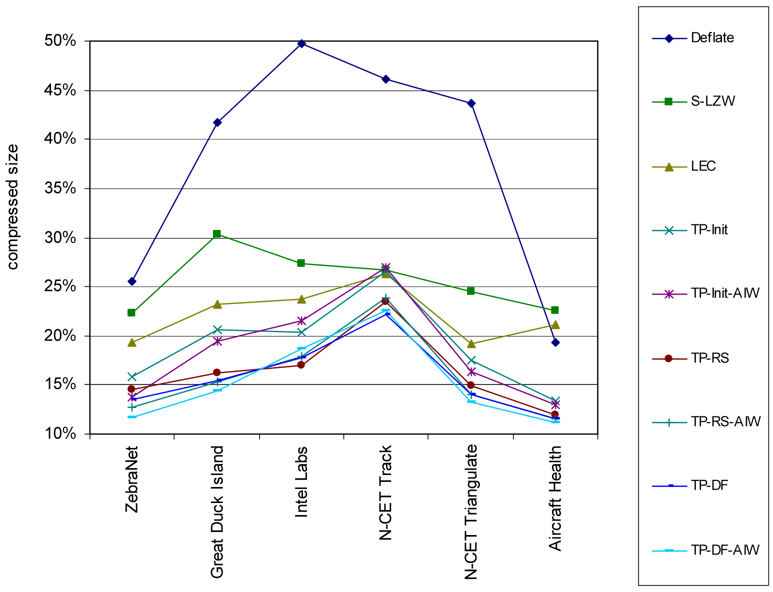
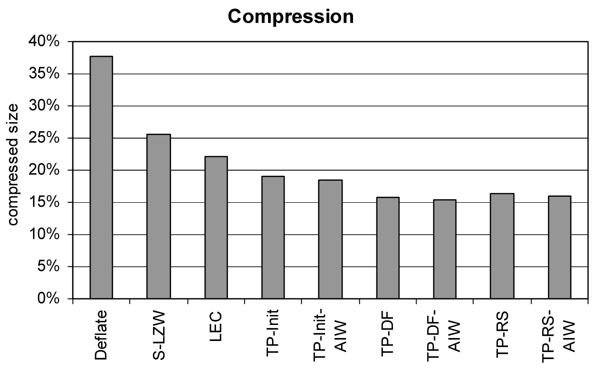


Fig. 11 Full compression results

Fig. 12 Compression summary



6.2 Accuracy

Since the TinyPack algorithms produce approximations of the frequencies of the values, a measure of accuracy can be calculated by comparing the lengths of the generated codes for each frame to the optimal code lengths determined by generating standard Huffman codes. Figure 13 shows the performance of the TinyPack and LEC algorithms compared to the performance of a theoretical optimal algorithm. Deflate and S-LZW both resulted in greater compressed sizes and are not shown here to allow for greater precision in the figure. It should be noted that while standard Huffman coding would produce optimal codes, the overhead for sending the new tree at every frame would cause the algorithm to perform much worse than any of the others. No algorithm currently exists which produces optimal codes with no overhead.

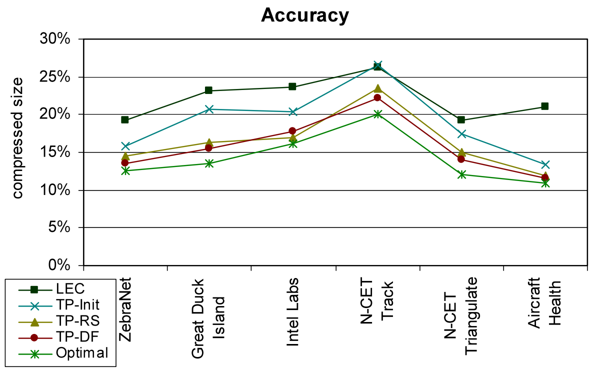


Fig. 13 Accuracy

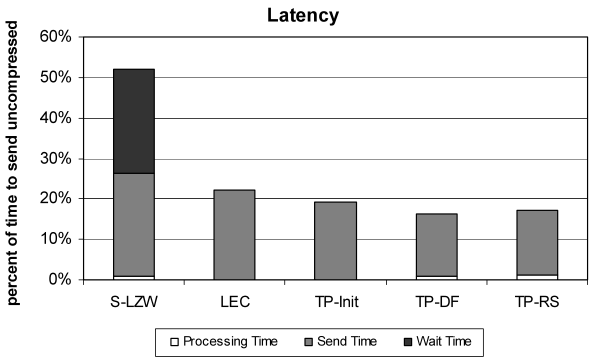


Fig. 14 Latency

The data in both Intel Labs and aircraft health remains fairly consistent throughout the entire dataset so the approximated codes almost reached the optimal level.

6.3 Latency

Sending the uncompressed data takes less time in processing but more time in transmission so the latency depends on the motes used. In general, however, processor speed is much faster than radio data rate for wireless sensors (for example, the Mica2 mote [14] has a 16 MHz processor and a 38.4 kbps high data rate radio). For the Mica2 motes, latency is decreased proportionally to the compressed size of the data. Thus, TinyPack has a decrease in latency of 80–85 % compared to uncompressed data. Latency was measured at the base station by querying the system clock at the beginning and end of each transmission and at the beginning of each nodes time window to determine the processing time. For S-LZW the nodes logged and averaged their own wait times and sent that data at the end of the experiment.

For comparison, the S-LZW algorithm was modified to send data as soon as possible and it was assumed packets were sent in a constant stream. Figure 14 shows the relative latencies scaled to the uncompressed data. In each version of TinyPack adding the all-is-well bit decreased the latency by less than half a percent so data for the all-is-well bit is not shown separately. Deflate is not shown since it requires collecting all of the data prior to compressing. Send time is directly proportional to

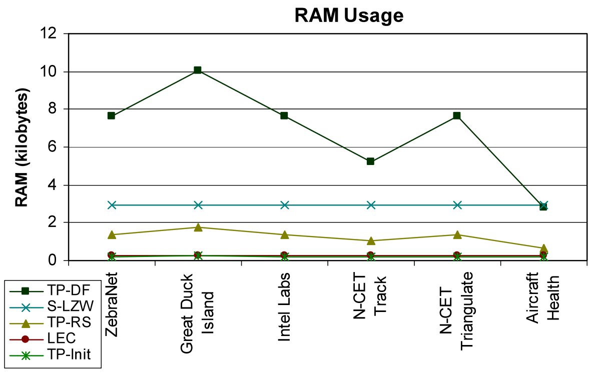


Fig. 15 RAM usage

compression (shown in Sect. 6.1) and processing time is directly proportional to the processor utilization (shown in Sect. 6.5).

6.4 RAM

The maximum amount of RAM utilized by each algorithm for each dataset is shown in Fig. 15. S-LZW is designed to work on any generic dataset and uses the same compressor for every value in a sensed vector making the RAM usage constant for S-LZW. As expected, TP-DF had the highest RAM usage because it stores the frequency tables; however, the RAM was still well within the limits of the Mica2, MicaZ, and most other sensors. LEC and TP-Init both use very little RAM since the codes are static and generated at runtime for each value.

6.5 Processor utilization

In order to measure processor utilization, the program counters on each sensor were accessed at the start and end of each simulation. For these simulations, the data was compressed and not transmitted to prevent the processor utilization from being affected by the compression ratio. Figure 16 shows the instruction count for each algorithm scaled to show the average instruction count per byte of uncompressed data. As with RAM, the static codes used in LEC and TP-Init cause the processor utilization to be very low. TP-DF and TP-RS required significantly higher processor time than the other algorithms; however, due to the nature of the sensor hardware, the savings in energy and latency from the reduced data size far outweigh the costs of higher processor utilization. The energy usage from processing is included in the results of the energy simulation in Fig. 17.

7 Experimental results using a sensor network simulator

Experiments were performed using TOSSIM [19], which simulates the open source TinyOS operating system that runs on many sensors. TOSSIM simulated Crossbow Technology's MicaZ motes [14] and was used to verify the experimental results as

Fig. 16 Processor utilization

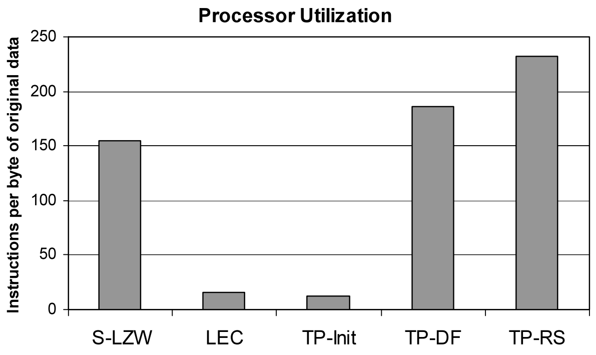
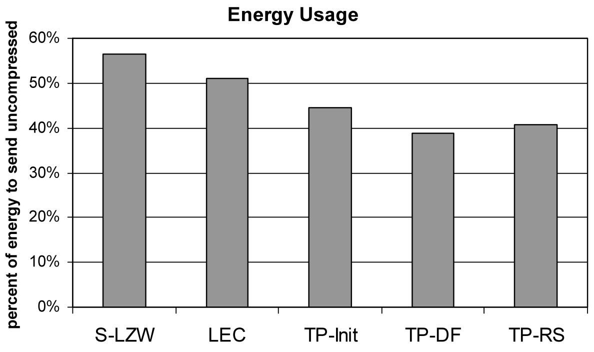


Fig. 17 Energy usage



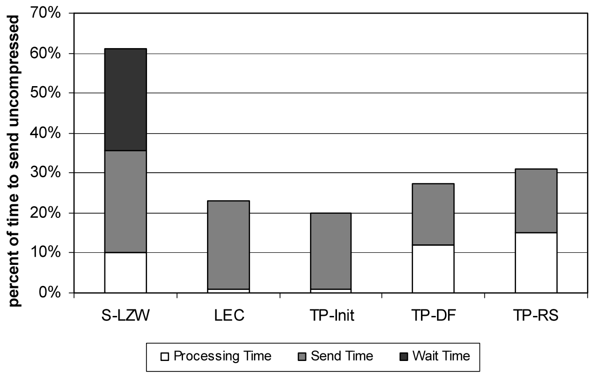
well as measure energy consumption and to test the algorithms under larger networks and different architectures. In addition to TOSSIM the PowerTOSSIM [20] simulator was used. PowerTOSSIM is built on top of TOSSIM and provided the capabilities of measuring simulated energy consumption and latency.

7.1 Energy usage

Energy consumed for compressing, writing to memory, and transmitting was measured using PowerTOSSIM. Results shown in Fig. 17 are again scaled to a percentage of the cost to send the data uncompressed and averaged over all the datasets. As with latency, the all-is-well bit in each case decreased the energy usage by less than half a percent. Energy usage data was not collected for the Deflate algorithm since it was included only as a compression benchmark and was not implemented in PowerTOSSIM. As can be seen by comparing Figs. 12 and 17, energy results closely matched the compression results since most energy is consumed while transmitting the data.

7.2 Latency in a multihop environment

Experiments were performed to show the effects of the algorithms in a multi-hop environment. Sensing nodes sent data to the base station through a varying length series of forwarding nodes. For sensors with a slower processor or faster radio, the



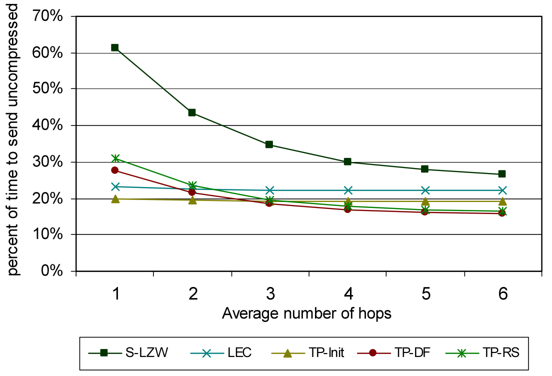


Fig. 18 Latency for high speed radio single and multi-hop

processor utilization becomes a greater factor, but in a multi-hop environment, the algorithms with the best compression ratio still outperform the others. Modifying the simulation to use a data rate of 2.5 Mbps radio like the Manchester-coded sensors in [21] generated the latency results shown in Fig. 18. The left graph shows the latency on a single sensor and the right graph shows how latency changes with the number of hops. As the average number of hops increases, latency approaches sending time since there is no additional processing needed when forwarding the compressed packets. After two or three hops the algorithms with the best compression ratio have the lowest end-to-end latency even for sensors with high speed radios.

8 Error detection and recovery

The first packet in a new frame is sent with uncompressed values. Each additional packet is sent using the delta (change) values. If the last value is repeated in the first packet of the next frame, the values can be compared to check for the presence of errors due to dropped packets or corrupted values in the packets.

For example, suppose a temperature sensor sensed values at 23, 25, 28, and 29 with a frame size of 4. The first frame contains [23, +2, +3, and +1]. Assuming packet corruption changed the +3 to −3, the receiver would read the values as 23, 25, 22, and 23. When the second frame was sent with 29 as the first value the receiver could see that an error had occurred since the last value (23) does not equal the first value of the next frame (29).

This successfully detects all single bit errors and single dropped packets; however, it is possible that multiple errors could cause the values of the compared packets to actually be equal although the errors existed. For example a $+2$ and a -2 could both be dropped. In this case the drops would be undetected.

Since the codes are dynamic, the chances of undetected error constantly changes but the codes in all cases were consistently distributed similarly to the static default codes so those were used for error analysis.

Experiments were conducted with errors generated assuming Poisson inter-arrival times and results were consistent with the following analysis.

8.1 Drop detection

For dropped packets, the probability of a subsequent error "correcting" the value and causing the errors to be undetected can be computed using a state diagram and transition matrix. The state number is defined as the difference between the value calculated at the receiver and the value transmitted by the sender. For example, state 3 represents that the receiver believes the value to be 3 greater than it really was and state 0 represents either no error or undetectable error. Since transitions can go from any state to any other state and the number of states is equal to twice the number of possible values, the diagram is far too complex to include. The probability of an error causing a transition from a state X to a state Y is

$$P(X, Y) = 2^{-2\lceil \log_2(|X-Y|+1)\rceil - 1}$$

Clearly $P(X, Y) = P(Y, X)$ so the probability of transitioning from X to Y and then from Y back to X is just $P(X, Y)^2$. The probability of a second error correcting the value and causing both errors to go undetected is represented by transitioning from the initial state 0 to any state X and back and is

$$\sum_{x=-\infty}^{\infty} P(0, X)^2 = \sum_{x=-\infty}^{\infty} 2^{-4\lceil \log_2(|X|+1)\rceil - 2} \approx .0357$$

Therefore the probability of two drops going undetected in a frame is roughly 3.57 %. Since most sensors send a vector of values at each sample the probability of detecting multiple errors from dropped packets is $(.0357)^{|V|}$ where $|V|$ is the vector size of the sample.

For example, the Intel Labs dataset contains 2.3 million samples with six values in each sample so $|V| = 6$. In the worst case there will be exactly two drops per frame. Assuming 10 % packet loss, there would be approximately 115,000 frames each containing two dropped packets. The chance of detecting every drop would be

$$\left(1 - (.0357)^6\right)^{115000} \approx 99.976\ \%$$

The worst case probabilities are shown for each of the datasets in Table 11.

Table 11 Probability of drop detection

Dataset	$\lvert V\rvert$	Frames	Probability
ZebraNet	6	284	99.9999 %
Great Duck Island	8	38226	>99.9999 %
Intel Labs	6	115123	99.9762 %
N-CET Track	4	23143	96.3106 %
N-CET Triangulate	6	11123	99.9977 %
Aircraft Health	2	22937	<0.00001 %

The aircraft health data has only two values per vector and so in the worst case, at 10 % drop rate, errors would undoubtedly go undetected. For such datasets, it would be effective to define a smaller frame size to reduce the probability of multiple errors occurring in the same frame or to send error detection packets in the middle of the frame instead of always sending them at the end.

8.2 Single bit error detection

Assuming the values occur with the probability expected by the default codes, the probability of a bit error occurring in the base (prefix) of a code can be determined by calculating the expected number of prefix and suffix bits in a code.

From Table 7 it can be seen that a code at level L has a prefix length $L+1$ and suffix length L. The count of nodes at that level is 2^L so the probability of a random sampled value being on that level is $2^{-(L+1)}$. Therefore the expected number of prefix bits $E(P)$ for an arbitrarily large set of possible values is:

$$E(P)=\sum_{L=0}^{\infty}\left(\frac{L+1}{2^{L+1}}\right)=\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\cdots$$

$$2E(P)-E(P)=2$$

Similarly, the expected number of suffix bits $E(S)$ is:

$$\begin{aligned}E(S)&=\sum_{L=0}^{\infty}\left(\frac{L}{2^{L+1}}\right)=\sum_{L=0}^{\infty}\left(\frac{L+1}{2^{L+1}}-\frac{1}{2^{L+1}}\right)\\&=E(P)-\sum_{L=0}^{\infty}\left(\frac{1}{2^{L+1}}\right)=1\end{aligned}$$

As the height of the tree approaches infinity, $E(P)$ approaches 2 and $E(S)$ approaches 1. The probability of a bit error occurring in the prefix for large trees approaches 66.67 %. Calculating for the case where the values can range from -127 to 127 gives 66.98 %. Such errors would change the expected length of the code and would either be detected at the end of the packet transmission or would cause the data to vary so greatly that the probability of a future error correcting the value is exponentially less than if the error was in the suffix.

Table 12 Error correction

Dataset	Errors	Average	>1 %
ZebraNet	57	0.18 %	2.5 %
Great Duck Island	7642	0.34 %	4.2 %
Intel Labs	23035	0.07 %	1.3 %
N-CET Track	4607	0.26 %	3.4 %
N-CET Triangulate	2231	0.19 %	2.9 %
Aircraft Health	4586	0.12 %	1.7 %

Suffix bit errors cause the error in value to change in the same way as dropped packets. Thus, the probabilities of errors going undetected are one third those of the dropped packets.

8.3 Correction

If the data is sent based on a sampling interval or if the packet headers contain sequence numbers, then the above error detection mechanisms can easily be used to reconstruct dropped or corrupted packets. In the case of a single dropped packet, the values dropped are equal to the difference between the calculated value at the receiver and the value of the error detection packet. For example, assume again that a temperature sensor sensed values at 23, 25, 28, and 29. The values encoded and transmitted would then be 23, +2, +3, and +1. Assume that the packet containing the +3 value was dropped and the calculated value at the receiver is $23 + 2 + 1 = 26$. At the end of the frame, the sender transmits the non-encoded real value of 29 as the error detection packet. Since $29 - 26 = 3$, the receiver can instantly calculate the missing value as +3. In the case of multiple dropped packets, the difference represents the total error over all drops. For consecutive drops, we simply divide the total error by the number of drops and assign that value to each missing packet. For non-consecutive drops, the values are scaled based on the ratio of the previous and next packet surrounding each missing packet.

We experimented using the same frame size of 512 and a 1 % Poisson distributed drop rate. Table 12 shows the average error compared to actual value of the dropped packet as well and the percentage of errors greater than 1 %.

9 Conclusions and future work

The TinyPack suite of protocols effectively compresses data while not introducing delays and even reduces latency compared to sending uncompressed data. TinyPack is effective on all sensor networks which use time-based sampling and is especially effective on systems with high granularity or low local variance.

TP-Init required the least RAM and by far the least processing time of all the TinyPack algorithms but resulted in the poorest compression. TP-DF achieved the greatest compression ratios, but required more RAM than the other methods. TP-RS compressed almost as well and required much less RAM. While TP-DF compressed most effectively, systems with low RAM would benefit from using TP-RS and systems with very low RAM or high cost for processor utilization could use TP-Init for best results.

While the focus of this paper has been lossless compression, TinyPack could be modified to continue sending change values of zero until the change exceeded some threshold. Additionally, packets could be dropped to indicate no change had occurred. In systems which could tolerate some rounding error or lossiness, this could dramatically increase the compression with a small degree of error.

In many applications sensors are not only temporally located but also spatially located (sensors sense data similar to that of a nearby sensor). It could prove effective to express the delta values as the change from the value of a nearby sensor instead of the change from previous value or some hybrid of the two.

References

1. Huffman, D.A.: A method for the construction of minimum-redundancy codes. In: Proceedings of the I.R.E. (1952)
2. Vitter, J.S.: Design and analysis of dynamic Huffman codes. J. ACM **34**(4), 825–845 (1987)
3. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977)
4. Arici, T., Gedik, B., Altunbasak, Y., Liu, L.: PINCO: a pipelined in-network compression scheme for data collection in wireless sensor networks. In: Proceedings of 12th International Conference on Computer Communications and Networks, October 2003
5. Petrovic, D., Shah, R.C., Ramchandran, K., Rabaey, J.: Data funneling: routing with aggregation and compression for wireless sensor networks. In: Proceedings of First IEEE International Workshop on Sensor Network Protocols and Applications, May 2003
6. Sadler, C., Martonosi, M.: Data compression algorithms for energy-constrained devices in delay tolerant networks. In: Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys) (2006)
7. Marcelloni, F., Vecchio, M.: An efficient lossless compression algorithm for tiny nodes of monitoring wireless sensor networks. Comput. J. **52**(8), 969–987 (2009)
8. Gandhi, S., Nath, S., Suri, S., Liu, J.: GAMPS: compressing multi sensor data by grouping and amplitude scaling. In: Proceedings of the 35th SIGMOD international Conference on Management of Data, New York, NY, pp. 771–784 (2009)
9. Mainwaring, A., Culler, D., Polastre, J., Szewczyk, R., Anderson, J.: Wireless sensor networks for habitat monitoring. In: WSNA'02: Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications, pp. 88–97. ACM, New York (2002)
10. Bodik, P., Hong, W., Guestrin, C., Madden, S., Paskin, M., Thibaux, R.: Intel Berkley Labs. (2004)
11. Zhang, P., Sadler, C.M., Lyon, S.A., Martonosi, M.: Hardware design experiences in ZebraNet. In: Proc. of the ACM Conf. on Embedded Networked Sensor Systems (SenSys) (2004)
12. Metzler, J.M., Linderman, M.H., Seversky, L.M.: N-CET: network-centric exploitation and tracking. In: MILCOM 2009—2009 IEEE Military Communications Conference, October. IEEE, New York (2009)
13. Zhao, X., Qian, T., Mei, G., Kwan, C., Zane, R., Walsh, C., Paing, T., Popovic, Z.: Active health monitoring of an aircraft wing with an embedded piezoelectric sensor/actuator network: II. Wireless approaches. Smart Mater. Struct. **16**(4), 1218–1225 (2007)
14. Crossbow Technology, Inc.: Mica2 and MicaZ Datasheets. http://www.xbow.com/ (2010)
15. Shannon, C.E.: A mathematical theory of communication. Bell Syst. Tech. J. **27**, 379–423, 623–656 (1948)
16. Madden, S., Franklin, M., Hellerstein, J., Hong, W.: TAG: a tiny aggregation service for ad-hoc sensor networks. In: Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02) (2002)
17. Sharaf, A., Beaver, J., Labrinidis, A., Chrysanthis, K.: Balancing energy efficiency and quality of aggregate data in sensor networks. VLDB J. **13**(4), 384–403 (2004)

18. Deligiannakis, A., Kotidis, Y., Roussopoulos, N.: Hierarchical in-network data aggregation with quality guarantees. In: Proceedings of EDBT Conference (2004)
19. Levis, P., Lee, N., Welsh, M., Culler, D.: TOSSIM: accurate and scalable simulation of entire TinyOS applications. In: Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys) (2003)
20. Shnayder, V., Hempstead, M., Chen, B., Allen, G.W., Welsh, M.: Simulating the power consumption of large-scale sensor network applications. In: Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys) (2004)
21. Chaimanonart, N., Suster, M., Ko, W., Young, D.: Two-channel data telemetry with remote RF powering for high-performance wireless MEMS strain sensing applications. In: 4th IEEE Conference on Sensors (2005)