TiredZebra: Exploring Gossip Protocols in Sensor Networks

CS 244B Final Project

Paul Crews
Stanford University
ptcrews@stanford.edu

Travis Lanham
Stanford University
tlanham@stanford.edu

ABSTRACT

Wireless sensor networks have extraordinary resource constraints that demand low power consumption in order to extend longevity. Designing distributed algorithms to work on low-power, resource-constrained devices is a challenging problem and requires a different approach than a typical distributed environment. In this paper, we examine one such algorithm, Trickle, in depth, experimentally measuring its behavior on real-world hardware and under simulated conditions. We also extend the algorithm to add an optimization that allows nodes to turn off their radios for a set period of time to conserve energy.

KEYWORDS

IoT, Distributed Systems, Sensor Networks, Networking

1 INTRODUCTION

Sensor networks have extreme resource constraints that require a distinct class of distributed algorithms and protocols designed for large numbers of lossy nodes with low-power requirements. The devices that make up these networks typically consist of small, embedded systems that are placed in remote locations and cannot connect to a centralized controller. The lack of a centralized control system necessitates peer-to-peer gossip protocols to propagate information, including code updates.

Power consumption is critical for these systems since their batteries cannot be replaced and thus battery life determines useful system life. Network radios on these systems are the largest consumers of energy and therefore must be the most optimized to reduce resource consumption. To compound the issue of propagation, wireless sensor nodes offer few synchrony guarantees and nodes constantly fail or experience packet loss.

However, several consensus algorithms have been designed to function in this restricted context. In this paper, we examined the Trickle algorithm, which is an algorithm for eventual consistency over unreliable wireless networks [3]. We present our own implementation of Trickle and measure its performance under different network conditions, both on real-world hardware and in a simulated environment. We also present an extension to Trickle, which makes different assumptions about the underlying network topology but enables some nodes in the network to sleep.

We first describe the background for the Trickle algorithm, including our proposed modification and its implications. We then discuss the methodology used to test Trickle in several network configurations and mediums, and evaluate the validity of claims made surrounding the recommended values for certain Trickle parameters. We also test our modified version of Trickle, Sleepy Trickle, under the same network conditions to see how the modified algorithm changes the overall power cost. We then present the results of our experiments, analyzing the results based on the expected behavior of the algorithm. Finally, we present our conclusions, and suggest that Sleepy Trickle is a viable extension to Trickle under certain network conditions.

2 BACKGROUND

2.1 Trickle Algorithm

The Trickle algorithm was initially developed to distribute code updates throughout a wirelss sensor network using a minimal number of packet transmissions while evenly distributing the transmit load over each node and preventing broadcast storms. One of the key insights during the development of Trickle was that transmission was much more expensive than radio reception; as a result, it was cheaper to leave radios on in the listen state if that reduced the overall number of transmissions. The algorithm was initially developed for TinyOS and associated projects, but is now incorporated in a number of other network standards [4].

The foundation of the algorithm is the proposition that nodes should remain quiet unless they see an update on the network and then should broadcast it out; after receiving an update, a node should advertise it minimally, exponentially backing off while making additional transmissions to preserve resources. The motivation behind this foundation is that nodes in range of each other will quickly propagate an update and then go into a standby mode with occasional transmissions until the next update is introduced. Trickle also sends all messages to a broadcast address which allows all nodes in range to hear an update without requiring pairwise connections for propagation.

The Trickle gossip algorithm operates on a series of time intervals where a random time is chosen from the current interval to decide whether or not to broadcast the current version. If the node has heard its current version broadcast by several other nodes in the current interval then it will not broadcast to avoid duplicate effort. If a node hears a version that is behind its current version, it will reset its interval to the minimum size then broadcast out the more recent version to bring the rest of the network up to speed.

The algorithm relies on several configurable variables to determine these time intervals. The time interval spans from $[0, T]$ from which it randomly chooses the time t to broadcast in the current time interval (note that t is not an absolute time, but rather an offset from the beginning of the time interval). Each node also has a counter c for tracking the number of updates it has heard in the interval; if the node hears k updates before it's transmission time t then it will not transmit. k is typically a small number (1 to 4) because if the update has already been heard several times then it is likely that it has spread throughout the network and gained saturation. For a sparse network, a lower k is desirable because even if a node hears another has an update, there might be nodes that were out of range and did not hear the transmission. After the interval T finishes, the counter c is reset and a new t transmission time is selected for the new interval.

However, in practice, the Trickle authors modified the original algorithm to select t from the interval $[\frac{T}{2}, T]$ instead of $[0, T]$. We also adopted this modification which aimed to address the "short listen" effect described by the authors where the lack of time synchronization results in nodes having offset intervals such that they would ordinarily suppress broadcasts but line up so a node broadcasts, then it's interval ends, then another node's interval begins and it broadcasts shortly after the first. This represents an adversarial case that can significantly increase the number of broadcasts and needlessly transmit messages. Taking the transmission time t from only the second half of the interval enforces a fixed waiting period for all nodes which provides an upper bound on the broadcast rate.

Another optimization the algorithm introduces is a dynamic configuration of the window timing interval. A large T interval has a low resource overhead while a small T propagates information through the network faster. Trickle responds dynamically by varying T within an interval $[T_l, T_h]$. After interval T_n, T_{n+1} is doubled to be $T_{n+1} = 2T_n$. An upper bound is established as T_h so doubling goes to that limit. If a node hears an update then it resets T to T_l and begins again. This is analogous to TCP window size in the exponential increase stage. Each interval that passes without an update results in a doubling of the prior interval length. Thus, as time passes and no new updates are heard, nodes will wake up less frequently.

2.2 Sleepy Trickle Extension

Our extension to the Trickle algorithm makes additional assumptions about the network topology. In particular, we assume that there is some set of router nodes R that form a connected graph and some disjoint set of leaf nodes L. Nodes in R are expected to be high-power, and nodes in L can be low-power. This network model is intended to resemble a common network configuration, where low-power weak nodes are connected to reliable, high-power nodes. Our modification exploits these properties to allow the lower-power nodes in L to sleep, while tuning Trickle parameters to increase the transmit load on the nodes in R.

The first modification we made was to increase the transmit load on the nodes in R, and we achieved this by increasing the Trickle k parameter for each node in R. As described in 2.1, the k parameter determines how many messages are required to suppress a transmission at a node. By increasing this parameter at some subset of nodes, we increase the broadcast load at these nodes and increase the probability that they transmit. Thus, even in a lossy network, this helps to suppress transmissions of nodes not in R - that is, the low-power leaf nodes.

In addition to varying the k parameter between nodes in R and nodes in L, we also allowed nodes in L to enter a sleep state. Our algorithm for entering sleep state is as follows; when a node in L has reached the maximum sized interval T_h and has not transmitted during the prior period, it is allowed to sleep for some interval T_s. Once waking from sleep, the node resumes normal operations with interval T_h. Note that this sleep state is equivalent to a node simply disconnecting from the network for the sleep interval T_s. The node is then allowed to sleep again after not transmitting for an interval of length T_h as before.

There are several interesting emergent properties of this modification. First, the normal Trickle algorithm runs on each of the nodes in R. Since these nodes are assumed to be normally connected, this ensures that updates are eventually propagated to every node in R, assuming disconnected nodes are eventually reconnected. A second interesting property of the modified algorithm is that this algorithm is equivalent to some subset of nodes in L disconnecting for time T_s in the standard Trickle implementation. Thus, many of the properties of standard Trickle is maintained in Sleepy Trickle, including at the sleepy nodes L. Finally, another interesting property is the long-term behavior of a disconnected set of nodes in L. Consider some subset L_d of L such that no node has received an update. Thus, since no external transmissions can be heard, $|L_d| - k$ nodes must remain awake for the next sleep period, as k must have transmitted. Note that this property holds regardless of sleep interval timing; thus, so long as one node in the $|L_d| - k$ awake nodes is eventually reconnected, the update will spread to the disconnected group. Additionally, since the selection of awake nodes is randomized based on when the t timer fires, there is a non-zero probability of a particular node remaining awake, indicating that so long as some node in L_d is eventually updated, the rest of the nodes are also eventually updated.

Although this modification preserves many of the guarantees of the original Trickle algorithm, there are several properties to Sleepy Trickle that reduce its efficiency. First, the update latency is significantly increased. Not only does it take substantially longer to update nodes in L, but in complex multi-hop networks, there can be additional propagation speedup in standard Trickle by sending updates through nodes in L. Since these nodes may be asleep, the total number of hops to update the nodes in R may be larger. In addition to latency, transmit efficiency is impacted. Although Sleepy Trickle attempts to synchronize sleep periods for nodes, an intrinsic property of Sleepy Trickle is that fewer nodes receive the update at around the same time. Thus, groups of nodes may receive the update in batches, which increases the number of overall transmissions (as some transmissions would have been suppressed by other nodes that were simultaneously updated). Finally, the guarantees of Trickle are more difficult to ensure in Sleepy Trickle.

3 IMPLEMENTATION

3.1 ZebraSim: Software Simulation

Our first contribution is the development of a software simulation environment for Trickle that we call ZebraSim (in reference to ZebraNet, a deployment of sensors to track zebra herds in Africa and an inspiration for the Trickle paper). The original Trickle paper discussed a software simulator used for validating the algorithm and parameter tuning (particularly for sparse network topologies). However, this simulator was not made available by the authors and from the description in their paper appears to be a fairly simple program that mathematically models the parameters.

ZebraSim introduces a flexible simulator that maintains state for each simulated node and provides flexibility for creating the node topology. Nodes can be placed randomly or in groups and with specified spacing constraints (groups or individuals can be close together to create a dense network or far apart for a sparse one that requires multiple hops).

ZebraSim exposes a basic interface for each node, namely a transmit function and a receive function. The receive function delivers a broadcast message to the node and invokes the node processing for it, consisting of incrementing c, and potentially setting up a message to be transmitted at the transmit time for the interval.

Each node can be configured with a k, T_l, T_h, location, optional travel distance, and optional packet loss.

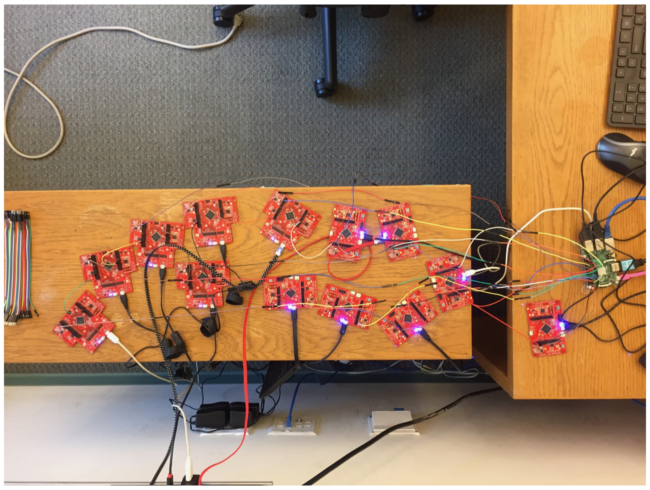

Figure 1: Imix boards with raspberry pi controller.

If travel distance is greater than 0 then the node will move position a random amount up to that distance in a random direction each interval (to model sensor networks that are geographically dynamic, for example, GPS tags for endangered animals). The packet loss parameter is a probability that a broadcast packet is not heard by the node (to introduce loss into the network).

We originally prototyped a multi-node simulator where nodes would be simulated with docker containers and networked together, however, this introduced noise into network measurements due to inter-container networking delay and was less reliable than a formal simulation.

3.2 Hardware Testbed

For our hardware testbed, we implemented Trickle and Sleepy Trickle on the Imix hardware platform running the Tock operating system [1]. Tock is a embedded operating system written in Rust and designed for low-power embedded devices, and represents a feature-rich test bed for low-power wireless research [2]. Only one modification was made to Tock itself, which involved adding back functionality for the hardware random number generator required by Trickle. The Imix board contains 64kB of RAM, a 40MHz Cortex-M processor, and a 2.4GHz IEEE 802.15.4 radio. This platform is an accurate representation for what kinds of low-power hardware Trickle was designed to run on, and captures the resource-constrained environment that both Trickle and Sleepy Trickle should run on. We used a total of 13 Imix boards to conduct our measurements, and connected GPIO pins to a Raspberry Pi to get accurate propagation measurements from a consistent internal clock as seen in Figure 1.

4 RESULTS

In this paper, we set out to examine two ideas. First, we were interested in the performance of Trickle under different network configurations, focusing on the claim that for dense networks, large values should be selected for k and T_l. Second, we were interested in the performance of Sleepy Trickle and how it compares to Trickle with regards to update latency and total transmission counts for low-power nodes. We highlight our results in the following subsections. First, in 4.1 we examine the behavior of Trickle with varying k and T_l values in a dense, single-hop network. In 4.2 we then test how the optimal k and T_l values perform in a multi-hop network environment. Finally, section 4.3 compares the performance of Sleepy Trickle to Trickle with regards to increased update latency and total transmission counts. We consider Sleepy Trickle a viable modification if it results in a moderate increase in update latency, and a substantial decrease in total transmission counts for the low-power leaf nodes. The results of our test confirm that Sleepy Trickle appears to meet both of these criteria.

4.1 Dense Single-Hop Network

Our first claim was that Trickle performs best in dense networks with a high k value and a high T_l value. In order to measure for performance, we measured the total time it took for an update to propagate through a network of connected nodes and the total transmission counts for the nodes.

We first tested this hypothesis on the hardware test bed. Using 13 total nodes, we used the configuration described in section 3.2 to measure the propagation delay and packet counts. For the single-hop network, we measured the latency for four different k and T_l value pairs as shown in Figure 2. We can see that the first test, for $k = 1, T_l = 1s$, had significant latency. We hypothesize that this is due to the hidden terminal problem and possible interference, delaying all nodes from immediately hearing the update. Once k is increased to 2 however, the delay dramatically drops; this makes sense, as each node must hear two transmissions to remain silent.

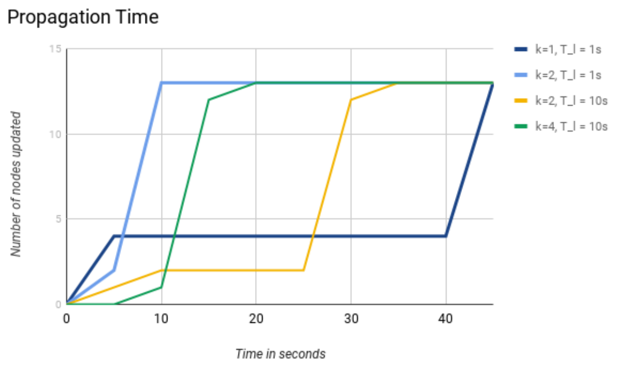

Figure 2: Propagation time for Single-Hop hardware nodes.

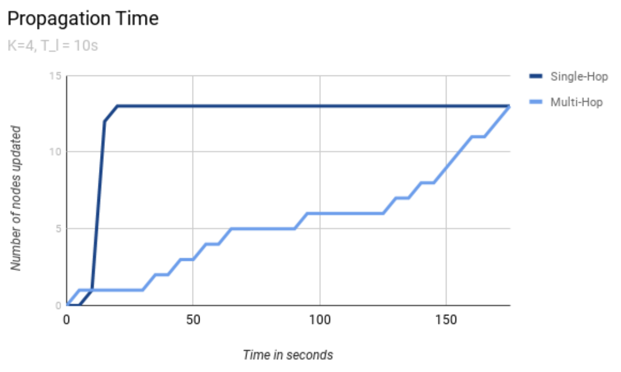

Figure 3: Propagation time for multi-hop vs single-hop.

In general, the tests seem to confirm the pattern that as k increases, the latency drops, while as T_l increases, the delay likewise increases. By increasing both values, this seems to minimize propagation delay while allowing for a quick increase in interval size. One potential problem in a dense single-hop network with a high k and a high T_l value is that the k value is approximately the total number of transmissions in an interval. This means that increases in k also increases the energy cost in a network, which can negatively impact the lifetime of battery powered nodes.

In our software simulation, we found similar patterns, although with a higher k value and higher T_l we had faster convergence which we attribute to no loss and a dense grouping that took only a few dozen intervals to fully propagate.

4.2 Multi-Hop Network

Although a high k and a high T_l value works well in dense single-hop networks, we wanted to examine how it performed in multi-hop networks. Since wireless network topologies can change dramatically over the course of a deployment, ensuring that this configuration also works for sparse multi-hop networks is crucial. To this end, we measured the end-to-end update propagation latency in ZebraSim and on the hardware testbed.

For the hardware testbed, we implemented the multi-hop simulation described in section 3.2. We used 13 nodes, each of which could only communicate with nodes whose MAC addresses differed by $+/-1$. This ensured worst-case performance, as each node could only receive updates from its immediate neighbor, creating a 13-hop long update chain. We then measured the end-to-end update time for $k = 2, T_l = 1s$ and $k = 4, T_l = 10s$ as shown in Figure 3. As the graphs show, there is a huge latency penalty for a large T_l in the multi-hop network. As a node must remain silent for the first half of an interval, when receiving an update with a large T_l value, the node remains silent for a significant amount of time. In our configuration, this creates substantial latency as each hop introduces a multi-second delay.

Although the high k and high T_l parameters perform well in dense single-hop networks, they perform particularly bad in large, multi-hop networks. However, the multi-hop network configuration tested here is most likely not indicative of real-world wireless networks, but it still demonstrates the behavior of Trickle under such conditions.

4.3 Sleepy Trickle

The final claim we tested was that our Sleepy Trickle modification decreases packet transmissions for leaf nodes while marginally increasing overall update latency. We tested this on the hardware testbed, measuring per-node packet transmissions while in steady state and while sending an update. We also measured the total update latency for all nodes, and compare these results to the standard Trickle algorithm in a dense single-hop network. We used a total of 13 nodes, 2 of which were the router nodes (nodes 0 and 1), while the remainder were sleepy nodes (nodes 2-12). The k-value for the router nodes (denoted k_r) we set to 4, while for the sleepy nodes we had $k_s = 1$. For the reference

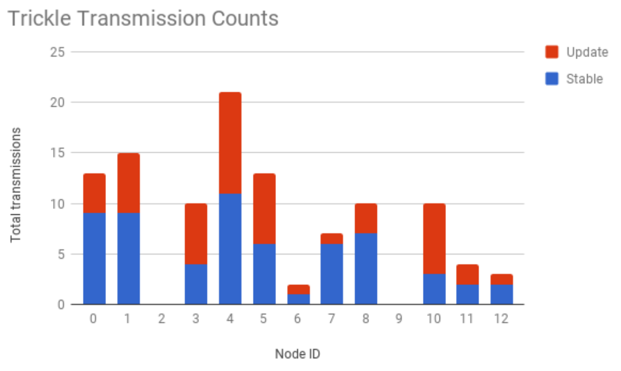

Figure 4: Normal Trickle transmission counts for $k = 4, T_l = 1s$.

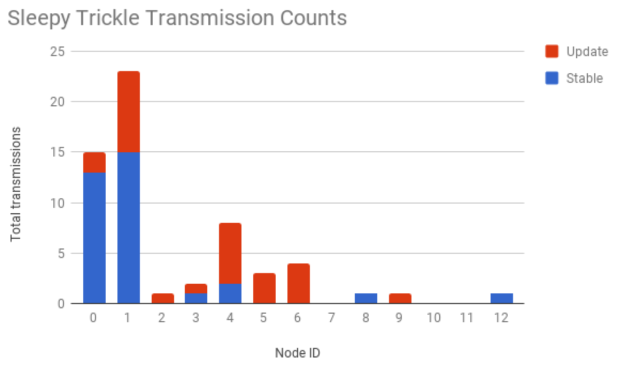

Figure 5: Sleepy Trickle transmission counts for $k_r = 4, k_s = 1, T_l = 1s$.

Trickle version, we set $k = 4$, and for all nodes we set $T_l = 1s$. We first let both Trickle and Sleepy Trickle stabilize, then measured 30 minutes of stable transmissions before initiating an update and measuring the total packet counts.

As Figures 4 and 5 show, the transmissions for Sleepy Trickle are dominated by the router nodes, and the net transmit count is much lower (59 transmissions) compared to standard Trickle (108 transmissions). For latency measurements, Figure 6 shows the difference in completion time for normal Trickle and Sleepy Trickle, with parameters $k = 2, T_l = 1s$ and $k_r = 4, k_s = 1, T_l = 1s$ respectively. Although Sleepy Trickle clearly increases the propagation time, we note that this is directly related to the amount of time nodes are asleep for (in this case, $T_s = 64s$). With these parameters, the increase in propagation time and the dramatic decrease

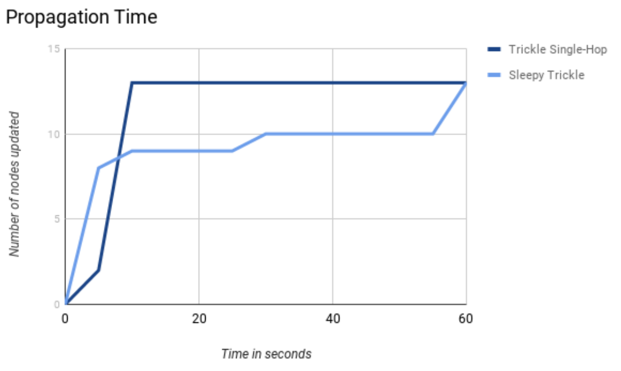

Figure 6: Trickle ($k = 2, T_l = 1s$) vs Sleepy Trickle ($k_r = 4, k_s = 1, T_l = 1s$) latency.

in transmissions by leaf nodes indicates that this modification is a viable extension of the Trickle protocol, and would enable more efficient deployment within specific network topologies.

5 CONCLUSIONS

With growing interest in internet of things embedded devices has come a resurgence of interest in low cost distributed protocols for information propagation. Trickle achieves the goal of fast code propagation through a lossy wireless sensor network while conserving power resources.

We further validate Trickle's performance, first with a high fidelity simulator, and second with a port to the Tock embedded operating system platform. We extend the original algorithm with Sleepy Trickle which takes concepts from traditional distributed networks (routing nodes) and apply them to the sensor network to achieve substantially lower transmission costs, at the expense of greater propagation delay.

REFERENCES

[1] github.com/lanhamt/sleepyzebra
[2] www.tockos.org/
[3] Levis, Philip, et al. "Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks." Proc. of the 1st USENIX/ACM Symp. on Networked Systems Design and Implementation. 2004.
[4] Hui, Jonathan W., and David Culler. "The dynamic behavior of a data dissemination protocol for network programming at scale." Proceedings of the 2nd international conference on Embedded networked sensor systems. ACM, 2004.