Self-Stabilizing Routing in Wireless Embedded Systems

Mikhail Nesterenko
Kent State University

Anish Arora
Ohio State University

Abstract

We address the problem of routing tree maintenance in a network of embedded sensors. We compare two routing protocols. One protocol periodically discards and reconstructs the routing tree. The other uses incremental routing tree construction. We analyze their advantages and disadvantages and describe experiments to which we subjected the protocols to compare their performance. We propose a protocol that combines the best features of both original protocols.

1 Introduction

The advances of technology have led to the emergence of an assortment of small devices capable of performing a variety of specific functions. One such emerging device is a wireless networked sensor [5]. Even though the functionality of a single networking sensor is limited, a collection of them working together can accomplish a variety of tasks ranging from treaty monitoring to inventory control and smart office spaces. Some of the potential applications require the deployment of a large number of such devices in adverse environments. In these conditions the problem or message routing becomes critical. The networked sensors in a region of 10^3 nodes have to be able to quickly and efficiently (a) obtain the routing information when the network is constructed (b) update the information in case of faults such as node failures and message losses. Such requirements call for self-stabilizing routing protocols.

In this paper we consider two self-stabilizing routing tree maintenance protocols: $\mathcal{A}$ and $\mathcal{B}$. The two protocols take opposing approaches to tree maintenance. Protocol $\mathcal{A}$ rebuilds the tree every timeout period completely discarding previous (possibly incorrect) data. Protocol $\mathcal{B}$ incrementally constructs the tree on the basis of the existing data. We analyze the performance of both protocols theoretically and experimentally and propose a protocol $\mathcal{C}$ that combines the advantages of $\mathcal{A}$ and $\mathcal{B}$. In related work Arora et al [1] consider high-atomicity self-stabilizing routing protocols.

As an experimental platform we selected prototype networked sensors running TinyOS [3]. Each network sensor has an embedded 4MHz microprocessor, 512 Bytes of RAM, 8KB of FLASH memory, an asynchronous short-range radio and an array of sensors. TinyOS is a lightweight (under 200Bytes of memory) highly configurable multithreaded operating system. TinyOS distribution contains a simulator TOSSIM that simulates networked sensors as Unix processes allowing the testing of a TinyOS program under a variety of simulated conditions.

The rest of the paper is organized as follows. We present abstract versions of $\mathcal{A}$ and $\mathcal{B}$ and analyze their performance in Section 2. We present our experimental results in Section 3. We conclude with a description of how $\mathcal{A}$ and $\mathcal{B}$ can be modified and how $\mathcal{B}$ can be used to extend $\mathcal{A}$ to provide adequate routing service for a region of networked sensors in Section 4.

2 Protocols $\mathcal{A}$ and $\mathcal{B}$

Model assumptions and protocol objective. A region consists of a number of processes $P_0, \cdots, P_N$. Each process has a unique identifier and can asynchronously broadcast messages to processes in its neighborhood. Channels are lossy, FIFO and bidirectional. We assume that the network is connected but otherwise arbitrary. One process P_0 is distinguished as a *base station*. The objective of the routing protocol is to enable each node to communicate (possibly indirectly) with the base station. Both protocols $\mathcal{A}$ and $\mathcal{B}$ maintain a distributed spanning tree rooted in the base station.

Protocol $\mathcal{A}$. A variant of $\mathcal{A}$ is described in [3]. A protocol for the base station (P_0) and the non-base processes ($P_i, i \neq 0$) are shown in Figures 1 and 2 respectively. In this protocol, to update the routing tree, the base station periodically broadcasts an `update` message (see $b1$). The period is an arbitrarily chosen value T. Each non-base process has variable *route* which stores the identifier of its parent. When a non-base process P_i receives an `update` ($o1$), it selects the sender to be its parent and rebroadcasts an `update`. Process P_i discards subsequent `update` messages for T time. The idea is that P_i selects its parent such that it lies on the route of the fastest message propagation from the base station. Ignoring updates after selecting the parent is done so that the value of *set* is not corrupted by "echo" messages and the propagation of the updates stops in the whole system before the next round of updates proceeds.

Protocol $\mathcal{B}$. Protocol $\mathcal{B}$ uses distance-vector (or distributed Bellman-Ford [2]) style routing. We use the hop count from the base station as the metric determininig the topology of the routing tree. The `update` carries the distance of its sender. The base station process (not shown) is the same as in $\mathcal{B}$ except `update` carries 0. The non-base process is shown in Figure 3. Process P_i selects its parent to be its neighbor with the smallest distance to the base station. Process P_i stores its distance and its parent's identifier in *distance* and *route* variables respectively. Periodically, P_i broadcasts its *distance* ($a2$). The period between broadcasts is T.

```
        process P_0
        const T : timeout length
        *[
b1 :        timeout(T) ⟶
                broadcast update
         ]
```

Figure 1: Base station process

```
        process P_i (i ≠ 0)
        const
            T : timeout length,
        var
            set : if parent is set,
            route : id of parent
        *[
o1 :        receive update from P_j ⟶
                if ¬set then
                    route := P_j,
                    set := true,
                    broadcast update
         []
o2 :        timeout(T) ⟶
                set := false
         ]
```

Figure 2: Non-base process of $\mathcal{A}$

We use route poisoning and triggered updates [4]. When P_i gets `update` from (1) a neighbor with smaller distance than P_i's parent or (2) P's parent with distance that differs from what P_i recorded ($a1$), then P_i updates its records and broadcasts an `update` with the new value of *distance*. If P_i does not get an update from its parent in $T + M$ time ($a3$) where M is the maximum propagation delay between neighbor processes in the system, it sets its *distance* to ∞ and broadcasts `update` with this value to notify the neighbors of the incorrect topology information. Note that M may have to be selected relatively large to account for message loss and retransmission by timeouts.

Performance evaluation. Protocol $\mathcal{A}$ is simple and it apparently works well in small networks. The metric that is used for topology tree constuction (fastest propagation time) is aggregate. That is, it includes, message delays, process load, transmission reliability etc, rather than just hop count. Discarding the

```
process P_i (i ≠ 0)
const
        T : timeout length,
        M : maximum message delay
var
        d : distance carried by update,
        distance : distance from base,
        route : id of parent,
        confirmed : if got update from parent
*[
a1 :    receive update(d) from P_j ⟶
            if (d + 1 < distance) ∨ ((route = P_j) ∧ (distance ≠ d + 1)) then
                distance := d + 1,
                route := P_j,
                confirmed := true,
                broadcast update(distance)
            else if (route = P_j) ∧ (distance = d + 1) then
                confirmed := true
  []
a2 :    timeout(T) ⟶
            if (distance < ∞) ∧ (route ≠ none) then
                broadcast update(distance)
  []
a3 :    timeout(T + M) ⟶
            if ¬confirmed then
                distance := ∞,
                route := none,
                broadcast update(distance),
            confirmed := false
  ]
```

Figure 3: Non-base process of $\mathcal{B}$

old tree and rebuilding it with every timeout makes $\mathcal{A}$ impervious to residual incorrect information. Message complexity of $\mathcal{A}$ is N for time period T, where N is the number of proceses in the system.

Let us consider the selection of timeout length for $\mathcal{A}$. Protocol $\mathcal{A}$ discards and reconstructs the routing tree every timeout period. During the construction the parent relation (created by *route* variable at each process) may not form a tree. The tree construction time depends on the size of the network. Thus, to maximize the tree availability the timeout length has to increase as the network size increases. Another reason to increase the timeout length with network size is $\mathcal{A}$'s lack of stabilization. We address this issue below. However, if the protocol has to respond promptly to the changes in system topology (node and link failures and recoveries) the timeout has to occur sufficiently often. We believe this contradiction limits the scalability of $\mathcal{A}$.

As shown $\mathcal{A}$ is not self-stabilizing: `update` can persist in the system between the broadcasts of the base station. This can be countered by appending the sequence numbers (SN) to `update`s. A sequence number can be bounded with bound B. The modified protocol works as follows. The base station increments (with wraparound) the sequence number of the update message each time the message is broadcast. Each non-base station process keeps the largest SN (modulo B) it received in variable *largest*. The process ignores updates with SN smaller than *largest*. SNs. If the process does not get an `update` carrying SN equal to or greater than *largest* for some time the value of *largest* is discarded.

We use hop count as a metric for the routing tree construction in $\mathcal{B}$. The constucted tree is stable in the sense that it remains fixed while the system topology does not change. Hence, the timeout period is affected only by the need for adequate response in case of topology changes and no explicit adjustment for network size is required.

In a stable state the message complexity of $\mathcal{B}$ is the same as $\mathcal{A}$. It takes up to $2N$ extra messages to adjust to a single process/link change of topology and it takes up to $2T + M$ time (discounting message propagation delay and assuming no message loss). We arrived at the figure by counting one message per process to propagate the information that the old tree configuration is incorrect (in case of process or link failure for example) and another message per process to construct the new tree configuration. Similarly it takes $T + M$ time for processes to determine that the old configuration is incorrect and propagate this information and another T to adjust it. Protocol $\mathcal{B}$ explicitly accounts for message delay (as well as message loss) in transmission which makes it easy to tune for the environment and application requirements. We discuss possible improvements to $\mathcal{B}$ in the conclusion.

3 Experiments

We coded $\mathcal{A}$ and $\mathcal{B}$ as application programs in Tiny OS and used TOSSIM to test their performance under various conditions. In particular we were interested in the protocols behavior under in the following cases: (1) common faults - process failures and message loss, (2) rare faults - such as multiple process failures or process failures (or other topology change) combined with massive message loss.

We ran the experiments for the systems of size 8, 64, and 254. We used the sytem of arbtirary topology. To generate the topology we assigned each process random coordinates on a plane and connected the processes within a certain distance of each other. The base station was placed in the center of the network. The networks were checked for connectivity. We ran two types of experiments. To model a rare massive fault we started the processes in an arbitary state (each process had random values assigned to *route* and *distance*) and measured the time it took the system to arrive at a state where *route*s of the processes in the system formed a tree. We ran the experiment with message loss rate of 0, 10, and 20 percents. We ran the experiment 100 times for each network size and each loss rate. To model common faults, we stared the system, waited until it stabilized and then stopped one arbitrary chosen process and measured the time it took the system to stabilize again. We ran the experiment 100 times for each network size and each loss rate. We set T and M for 2 and 8 seconds respectively.

Interestingly, $\mathcal{A}$ outperformed $\mathcal{B}$ under all conditions in both experiments. Our conjecture is that $\mathcal{A}$ reaches its scalability limits with larger network sizes or smaller timeout values. However, the limitations of TOSSIM did not allow us to test our conjecture.

4 Comparing and Combining $\mathcal{A}$ and $\mathcal{B}$

Protocol $\mathcal{A}$ constructs the routing tree fast and with relatively low message overhead. However, the availability of the routing tree decreases with system size which limits its scalability. Procol $\mathcal{B}$ does not have this limitation. Its Unfortunately, $\mathcal{B}$ has counting to infinity and route loop problems to which all distance vector protocols are prone. In addition $\mathcal{B}$ requires a relatively large number of messages for topology corrections. The problems can be mitigated (but not eliminated) by split horizon and hold-down techniques [4].

To achieve greater scalability we propose to combine $\mathcal{B}$ and $\mathcal{A}$. Protocol $\mathcal{A}$ quickly stabilizes the regions small enough for $\mathcal{A}$ to perform adequately; using the services of $\mathcal{A}$, $\mathcal{B}$ will maintain the global routing meta-topology of the larger region. In conclusion we a sketch the combined protocol $\mathcal{C}$.

The protocol $\mathcal{C}$ works as follows. Each process has two modes of operation – head and regular. Regular process behaves just like a process in $\mathcal{A}$ (with a small exception explained later). A head process periodically rebroadcasts an update message. Thus, essentially a head process behaves like a base station in $\mathcal{A}$. To complete the desciption we demonstrate how to (a) limit the propagation of updates to finite size (b) select head processes so that the set of head processes is relatively stable. For the former, an update message carries the hop count it traveled. When the hop count reaches a certain limit the update is discarded.

Thus, the propagation stops. For the latter we select the head processes by levels. The 0-th level head process is the base station. The next level head processes are on the outskirts of the propagation of updates from the previous level head processes. Since the routing tree that $\mathcal{A}$ construcs may be different every time, the head processes have to average the information carried by updates for serveral rounds. To keep track of levels, an update message carries the level number.

References

[1] A. Arora, M. Gouda, and T. Herman. Composite routing protocols. In *SPDP: 2nd IEEE Symposium on Parallel and Distributed Processing*. ACM Special Interest Group on Computer Architecture (SIGARCH), and IEEE Computer Society, 1990.

[2] L. R. Ford and D. R. Fulherson. *Flows in Networks*. Princeton University Press, Princeton, 1962.

[3] Jason Hill, Robert Szewczyk, Alec Woo, David Culler, Seth Hollar, and Kristofer Pister. System architecture directions for networked sensors. *ACM SIGPLAN Notices*, 35(11):93–104, November 2000.

[4] Christian Huitema. *Routing in the Internet*. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 2000.

[5] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S. J. Pister. Smart dust: Communicating with a cubic-millimeter computer. *Computer*, 34(1):44–51, January 2001.