Stabilizing Information Dissemination in Wireless Sensor Networks

Sain Saginbekov and Arshad Jhumka
Department of Computer Science, University of Warwick, Coventry, CV4 7AL, UK

Abstract—One important component of network reprogramming is code dissemination, when the new program code is distributed to the relevant nodes. Very few information dissemination protocols, if any, tolerate transient data faults, i.e., faults that corrupt the memory state of nodes. We address this limitation by proposing a novel protocol, called Repair, that transforms any fault-intolerant information dissemination protocol into a stabilizing protocol where, eventually, all nodes obtain the updated code. We conduct simulations to show the performance of Repair, and we integrate Repair with Varuna, and our results show that Repair induces low overhead on Varuna, and causes all nodes to receive the new code. Our main contribution is the first corrector protocol that stabilizes information dissemination in the presence of transient faults.

Keywords: Information dissemination; Transient faults; Stabilization; Wireless sensor networks; Composition

1. Introduction

Wireless sensor networks (WSNs) have enabled the deployment of several novel classes of applications, such as monitoring and tracking. However, to be useful, they need to operate unattended for long periods of time. However, this operational mode places several requirements on the network applications, but mainly that the applications are able to adapt to changing conditions. Given that WSNs are often deployed in hostile environments, human intervention is impossible. Thus, over-the-air reprogramming becomes a fundamental activity.

A network reprogramming protocol consists of several specific components: (i) a component that decides whether a complete code needs to be sent or only an update, (ii) information dissemination component, and (iii) reliability components [1]. The reliability component is to ensure that nodes receive all the parts of the code update that may be lost due to collisions. In this paper, we focus on the information dissemination aspect of network reprogramming.

Several information dissemination protocols have been proposed as part of network reprogramming protocols [2], [3], [4], [5]. However, to the best of our knowledge, none of them is fault-tolerant, i.e., none of them tolerate transient faults that corrupt the memory state of the nodes, which are known to occur in WSNs [6], [7], [8]. Such transient faults are also called soft errors. Given that several information dissemination protocols work by advertising the metadata of the new code, e.g., [2], [3], [9], any corruption of the metadata can, in the worst case, lead to the network nodes having stale code. Thus, it is important to have make these information dissemination protocols fault-tolerant. One trivial way of addressing this problem is to have the base station to periodically initiate the dissemination of the new code. However, this is a very expensive process, not suitable for WSNs.

In this work, instead of proposing a new fault-tolerant information dissemination protocol for WSNs, we propose a corrector protocol, called Repair, that, when integrated with a fault-intolerant information dissemination protocol, transforms the protocol into a stabilizing [10] fault-tolerant dissemination protocol, i.e, a protocol that guarantees that, eventually, all nodes will download the correct code. The novelty of our approach is that Repair enables the deployment of several stabilizing fault-tolerant information dissemination protocols. The corrector protocol, Repair, is triggered only when an erroneous state exists in the network, thus no overhead is incurred when no transient failures occur.

Contributions: In this context, we make the following contributions:

- We present a corrector protocol called Repair that, when integrated with any fault-intolerant information dissemination protocol, generates a corresponding stabilising fault-tolerant information dissemination protocol, and we prove its correctness.
- We run simulation experiments on Repair using TOSSIM [11], and show the performance of Repair, especially its locality property.
- We present a case study where we compose Repair with an existing dissemination algorithm, namely Varuna [3]. We show that our protocol (Repair) induces very little overhead over the Varuna in presence of transient faults. Further, Varuna, when executed in presence of even a single transient fault resulted in all the nodes downloading the wrong code. In contrast, when running Varuna with Repair, all the nodes eventually obtained the new code.

The paper is structured as follows: In Section 2, we present an overview of related work. In Section3, we present the system and fault models assumed in the paper. In Section 4, we present a corrector algorithm that stabilises the information dissemination of code updates. We present the simulation setup to evaluate the performance of the proposed algorithm in Section 5. The simulation results are presented in Section 6. In Section 7, we present a case
study where we integrate the algorithm with an existing information dissemination protocol to show the viability of our approach. We conclude the paper in Section 8, and present some avenues for future work.

2. Related Work

Information Dissemination: There exist several dissemination protocols which update nodes’ codes to new ones. While some of the protocols deliver complete binary image of the code like [12], [13], [9], [4], some deliver only the difference between the new code and the old code [14], [15]. Also there exist protocols which deliver tasks [16], network parameters [17], and queries [18].

In XNP[12], the base station broadcasts the code image to the nodes which are in its coverage range. The nodes outside of the range cannot receive the code image. The protocol proposed in MOAP[13] is a multihop dissemination protocol that can deliver code images to nodes that are several hops away from the base station. Each node forwards the code image further after receiving the complete code image. Deluge [9], allows large data transmission by fragmenting data into fixed-size pages. It also supports pipelining page transmission to make dissemination faster. Unlike MOAP, nodes in Deluge should not wait for complete code image before forwarding it. Authors of MNP [4] proposes another protocol like Deluge, which fragments the code image and uses pipelining mechanism. However, unlike Deluge, MNP selects the sender of the code such that there is only one sender at a time in a neighbourhood. Sender selection reduces collision and hidden terminal problem. Also, in MNP, some of the nodes can go to sleep mode to save energy whenever there is no data to receive and transmit.

Because of the feature of a wireless sensor network, such as transient link failures and node mobility, not all nodes update their code to the newest one during dissemination phase. The Trickle algorithm [2] addresses this problem by using a “polite gossip” policy. In Trickle, every node broadcasts advertisement messages, a metadata that includes version number of the code, at most once per period given between \[\tau/2, \tau\]. If a node hears more than \(k\) identical metadata before it transmits, it suppresses its broadcast and doubles the value of \(\tau\) up to \(\tau_h\), which is upper bound for \(\tau\). If it hears different metadata, \(\tau\) becomes \(\tau_l\), which is lower bound for \(\tau\). Varuna [3] is another protocol which supports code update maintenance. This protocol saves energy in steady phase, the phase where no dissemination is being done. Unlike Trickle, where there is a linear increase of energy consumption, energy consumption in Varuna is constant in steady phase. To achieve constant energy consumption in steady phase, nodes in Varuna send advertisement messages only when there is a change in neighbourhood topology or metadata since its last advertisement transmission. In Varuna, a node detects a fault when its version is less than that of the sender. These protocols do not consider transient memory faults, which may lead to a protocol to work incorrectly. For example, a node with new code may download old code if such a fault occurs. To the best of our knowledge, the work presented in this paper is the first to address fault-tolerant information dissemination. Fault Tolerance: In [19], it has been shown that a class of components, known as correctors, is sufficient to design non-masking fault tolerance. Thus, stabilisation, which is a special type of non-masking fault tolerance, is achieved by adding corrector mechanisms to a program, thereby transforming the program into a non-masking fault-tolerant one. Correctors are components that enforce a given predicate on program executions, whenever the predicate has been violated. The area of self-stabilization is mature, and several stabilizing algorithms exist [10].

3. Models: System and Faults

Graphs and networks: We define a wireless sensor node, or node, as a computing device equipped with a wireless interface and associated with a unique identifier. A node can communicate with a set of other nodes that lie at a certain distance from it. Generally, communication in wireless networks is typically modelled with a circular communication range centred on the node. However, we do not assume that all nodes have the same communication range and we do not assume that the range is circular. In this model, a node is thought as able to exchange data with all devices within its communication range.

A wireless sensor network is a collection of wireless sensor nodes and is modelled as a directed graph \(G = (V, E)\) where \(V\) is a set of \(N = |V|\) wireless sensor nodes and \(E\) is a set of edges or links, each link being a pair of distinct nodes. A node \(n \in V\) is said to be a 1-hop neighbour of a node \(m \in V\) iff \((m, n) \in E\), i.e., \(m\) can send a message to \(n\). Observe that communication need not be symmetric, i.e., if \(m\) can send a message to \(n\), \(n\) may not be able to do so. We denote by \(M\), the set of \(m\)’s 1-hop neighbours (or neighbours, for short). We say that two nodes \(m\) and \(n\) can collide at node \(p\) if \((p \in M) \land (p \in N)\).

Faults: A fault model stipulates the way programs may fail. We consider transient data faults that corrupt the state of the program by artificially corrupting the values held by variables. These faults are also known as soft errors.

Definition 1 (\(d\)-local algorithm): Given a network \(G = (V, E)\), a problem specification \(\Psi\) for \(G\), and an algorithm \(A\) that solves \(\Psi\), Algorithm \(A\) is said to be \(d\)-local if a node \(n \in V\) executes a transition that requires the state of its \(d\)-hop neighbourhood to be queried.


In this section, we present a corrector protocol, called Repair. When Repair is composed with a fault-intolerant protocol, called
dissemination protocol $\Sigma$, the resulting protocol ($\text{Repair}|\Sigma$) (pronounced Repair composed with $\Sigma$) is a stabilizing dissemination protocol, which guarantees that all nodes eventually download the correct code.

### 4.1 The Repair Protocol

*Repair*, shown in Figure 1, works with all reprogramming dissemination protocols that enable the detection of a fault. Specifically, Repair is triggered only when an erroneous state is detected due to a transient fault. Thus, any dissemination protocol that can interface with Repair must have enough state information to determine when a state is erroneous and, to the best of our knowledge, all current information dissemination protocols enable this. When a dissemination protocol detects a fault, then *Repair* is executed. In *Repair*, only corrupted nodes upgrade their code, avoiding unnecessary code updates.

*Repair* uses six special types of data packets (we call them *Repair* packets), in addition to dissemination packets used in the dissemination protocol:

- **Prob**: It contains code metadata and it is used to ask a neighbouring node to correct a fault.
- **Check**: It is used to request neighbouring nodes’ code metadata.
- **Rep**: It contains a node’s metadata and is used to reply to **Check** packet.
- **OK**: It is used to release some nodes from the correction process.
- **Cor**: It contains the correct metadata and is used to inform nodes about the correct metadata.
- **Hello**: It contains the correct metadata and is used to inform nodes that it has correct code.

Informally, *Repair* works as follows (The detailed algorithm is in Figure 3): When a node $n_1$ detects a fault after communicating with node $n_2$, it attempts to correct the fault. A **Prob** packet is sent to indicate a problem. If the fault cannot be corrected with $n_2$, then **Check** packets are broadcast, creating a correction tree, rooted at the node ($n_1$) that detected the fault. The leaf nodes of the tree responds to **Check** packets by sending **Rep** packets. If any of the leaf nodes detects a fault, it will spawn a subtree, within the main correction tree. Once a region in the network is reached where there is no fault, then no more subtree is spawned. This means that a node’s neighbourhood have the same code version, with the version being presumed correct, since the probability of identical corruption of the metadata is very low. Then, this node responds through a **Rep** packet, and its subtree “disappears”. Any node sending a **Rep** packet will cause its subtree to “disappear”. Ultimately, **Cor** or **Hello** packet will be issued, with node $n_1$ getting the correct code. Notice that, since Repair ensures $f$-locality, the correction tree will be on depth $f$.

**Fig. 1:** The state machine. Two states in dashed area are the states of any dissemination protocol. $f1=\text{TRUE}$ if sender of **OK** packet is the node which sent $P$, $f2=\text{TRUE}$ if Sender.Vers=Receiver.Vers, $f3=\text{TRUE}$ if all received metadata are the same.

#### 4.2 Proof of Correctness of Repair

**Lemma 1**: (f-local stabilization): Given a network $G = (V,E)$, fault model $F$, and a set $S$ of corrupted nodes, with the diameter of the corrupted area being $f$. Then, *Repair* guarantees that, eventually, all nodes in $S$ will have the correct code.

**Proof**: Assume that a node $n_f$ has detected a fault and wants to correct it. Given the diameter of the corrupted area is $f$, i.e., the maximum distance between two corrupted nodes in an area is $f$ hops, we assume that all nodes on the path between $n_{f-1}$ and $n_0$, namely $n_{f-2}, n_{f-3}, \ldots, n_0$ are corrupted. Node $n_0$ has neighbours with correct versions since corrupted area is of diameter $f$. We will prove by induction that node $n_f$ will eventually get the correct code.

**Base case**: Since all neighbouring nodes of $n_0$ have correct metadata, according to *Repair*, $n_0$ will receive **Rep** packets with all equal metadata. Then $n_0$ requests the correct code from one of the neighbours and goes to Disseminate state to download the correct code.

**Inductive hypothesis**: Assume that a node $n_i$, where $0 < i < f-1$, eventually receives **Hello** packet and updates its code from node $n_{i-1}$.

**Inductive step**: We need to prove that a node $n_{i+1}$, a neighbour node of $n_i$, eventually receives **Hello** packet and updates its code.

We know that in our protocol every node broadcasts **Hello** packet periodically up to $h$ times after receiving **Hello** packet or updating its code. Now there are two cases that can happen after $n_i$ receives **Hello** packet and updates its code.

- **Case 1**: $n_{i+1}$ receives a **Hello** packet and updates its code from $n_i$, which proves the inductive step.
- **Case 2**: Because of message losses, $n_{i+1}$ will not receive **Hello** packet from $n_i$. In this case, $n_{i+1}$ waits $Wait\_Time$ and goes to PS state and starts to receive and send $P$ packets. Eventually, $n_{i+1}$ or a neighbour
node of $n_i + 1$ will detect the fault, and executes Repair again, this time the affected area will be of size $(f - i)$, assuming no further fault has occurred. Assuming that the number of message losses is finite, eventually, $n_i + 1$ will get a Hello packet. The first case will be eventually true.

**Theorem 1 (Correctness of Repair):** Given a network $G = (V, E)$, transient fault model $F$, a $F$-intolerant information dissemination protocol $\Sigma$ for a specification $\sigma$. Then, Repair is a corrector component of $\Sigma$ for $\sigma$.

**Proof:** The proof follows from Lemma 1, and from the fact that Repair is only triggered when a fault is detected.

Observe also that Repair is self-correcting, i.e., if a transient fault occurs in Repair when Repair is executing, leading to a node downloading the wrong code. This fault will eventually be detected, and Repair will be executed again to correct the fault. Put otherwise, Repair is a corrector component for Repair itself.

Fig. 2: Variables of Repair algorithm.

Fig. 3: Repair Algorithm.

5. Experimental Setup

We perform TOSSIM[11] simulations on a 20x20 grid network to evaluate Repair. We set the distance between neighbouring nodes to 10 feet. Each node has a communication radius of around 30 feet. Network topology with asymmetric links is constructed by a tool given on tinyos.net. Each node is given a noise model from a heavy-meyer noise trace file.

Parameter values used in our simulation are given in Table 1. Some of the parameter values depend on other
parameters. For example, $WaitRep\_Time$ is the time for waiting for $Rep$ packets after broadcasting $Check$ packet. So, $WaitRep\_Time \geq SendCheck + SendRep$. $Wait\_Time$ should be set according to the code size and the size of the network. If the network and code size is large, this time should be large enough to allow neighbouring nodes to correct their code and forward it. Usually nodes enter $Temp$ state from $Wait$ state where it waits less time. The only case when a node waits $Wait\_Time$ is when there is a packet loss. $Temp\_Time$ time does not depend on other parameters. The value of $t$ should be small because a node waits a maximum $SendProb$ time to receive all possible $Prob$ packets. The values of $h$ and $p$ can be set to any values.

In our simulations, each node periodically broadcasts $P$ packet, with period randomly selected between $[0, U]$ at the start. We simulated two scenarios: (i) we incremented the number of corrupted nodes per circular area, which has diameter of 60 feet, and (ii) we kept the number of corrupted nodes to 5 and increased the size of a given (square) area, i.e., decrease the fault density. In both scenarios, the corrupted nodes were selected randomly in the given area. We then counted (i) the number of packets, (ii) the number of involved nodes, i.e., nodes that sent a $Repair$ packet, and (iii) the number of nodes which changed their states to $Wait$ and/or $WaitRep$ states. For each given number of corrupted nodes in the first scenario and for each length of square area in the second scenario, we ran simulations 5 times and computed the min, average and max values.

6. Simulation Results

In this section, we present two metrics to show the locality property of $Repair$, namely (i) number of nodes executing $Repair$, and (ii) number of packets sent.

**Number of nodes:** From Figure 4(a), we observe that, on average, the number of nodes executing the protocol varies linearly with the number of corrupted nodes. Given that the number of nodes involved is much less than the size of the network, it indicates that the number of nodes involved is proportional to the size of the corrupted area. Further, in Figure 4(b), we observe that, when 5 transient faults were injected, the number of nodes executing the protocol becomes almost constant, on the average. This is because, with decreasing fault density, the transient faults appear as single independent faults, with each of them involving a similar number of nodes, i.e., since the area increases, the chance of the 5 faulty nodes being neighbours is very low. These two observations support the fact that $Repair$ in $f$-local, where $f$ is the diameter of the fault-affected area.

**Number of packets:** We observe a similar trend as in Figure 5, further supporting the $f$-locality property of $Repair$.

7. Composition of $Repair$ with Varuna

In this section, we discuss the composition of $Repair$ with an existing information dissemination protocol, namely Varuna [3]. The reason for choosing Varuna is that it is one of the latest information dissemination protocol that have been proposed.

As mentioned before, $Repair$ is triggered by the detection of a fault. In Varuna, such a detection is enabled by one of the following conditions: (i) two nodes’ metadata (i.e., version number) are corrupted in such a way that the difference in versions is greater than 1, and (ii) the receiver of an advertisement message finds that its version is bigger than the advertised one and, at the same time, the sender exists in its neighbourhood table.

We simulated the composite protocol of Varuna and $Repair$ in TOSSIM. All nodes, except faulty nodes, are booted in the first minute. Faulty nodes are located at the center of the network. A packet with new version number is injected at 2 minutes. We simulated three faulty scenarios: (i) with 1 fault, (ii) with 4 faults and (iii) with 7 faults. For each scenario, we booted the faulty nodes (i) 30 seconds, (ii) 45 seconds and (iii) 60 seconds after injecting the new code, so that only a proportion of nodes had the new code version. We are specifically interested in (i) the overhead induced by $Repair$ on the performance of Varuna and (ii) the number of nodes with correct code at a given time. We simulated Varuna in conditions similar to those detailed in Section 5. Further, the values for Varuna-specific parameters are: DISS-RAND=2 sec, ADV-RAND=2 sec, $\tau$=8 sec, $T_{MOODY}$=1 min. For reasons of space, we only present a sample of the results obtained.

**Performance of $Repair$:** As can be observed in Figures 7 and 8, in all cases, injecting transient faults in the network during Varuna only execution causes the whole network to disseminate stale code. On the other hand, when Varuna is composed with $Repair$, every node downloads the correct code.

**Packet Overhead:** In Figure 6(a), it can be seen that the packets overhead induced by $Repair$ on Varuna is low. Specifically, with 7 faulty nodes, the packet overhead is less than 3%. From Figure 5, it can be observed that the number of packets will increase linearly with increasing number of corrupted nodes. The reason for the linear increase (as opposed to a constant value) is that the fault density increases when more corrupted appear at the centre of the network (condition under which we simulated the composite protocol).

**Temporal Overhead:** In Figure 6(b), it can be observed that the whole network receives the new code in approximately 80 seconds, after the new code has been injected into the network. Further, it can be observed that, when there are faulty nodes in the network, the time for the whole network to receive the correct code is approximately 80 seconds. Thus, there is almost no temporal overhead induced by $Repair$ on Varuna, highlighting the composable nature of $Repair$. This is so because $Repair$ executes in parallel with Varuna, and also corrects only corrupted nodes.
Table 1: Parameters for the simulation

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait_Time</td>
<td>50 sec</td>
</tr>
<tr>
<td>SendRep</td>
<td>2 sec</td>
</tr>
<tr>
<td>SendHello</td>
<td>1.5 sec</td>
</tr>
<tr>
<td>Temp_Time</td>
<td>30 sec</td>
</tr>
<tr>
<td>SendCheck</td>
<td>2 sec</td>
</tr>
<tr>
<td>WaitRep_Time</td>
<td>4 sec</td>
</tr>
<tr>
<td>SendProb</td>
<td>1 sec</td>
</tr>
<tr>
<td>t</td>
<td>0.2 sec</td>
</tr>
<tr>
<td>PeriodProb</td>
<td>7 sec</td>
</tr>
<tr>
<td>p</td>
<td>5</td>
</tr>
<tr>
<td>U</td>
<td>60 sec</td>
</tr>
</tbody>
</table>

8. Conclusion and Further Work

We have presented Repair, which when integrated with a fault-intolerant information dissemination protocol, transforms it into a stabilizing fault-tolerant one. We have shown the performance of Repair, and also when it was integrated with Varuna, one of the most recent information dissemination protocol. In presence of faults, Varuna causes all nodes to download the wrong code, while the composite protocol of Varuna and Repair ensured that all nodes eventually download the new code, while incurring minimal overhead.

References

Fig. 7: Varuna and Varuna Repair: (a) 4 faulty nodes are booted 180 seconds after new code injection (at 5 minutes). (b) 7 faulty nodes are booted 30 seconds after new code injection (at 2 minutes 30 seconds).

Fig. 8: Varuna and Varuna Repair: (a) 7 faulty nodes are booted 45 seconds after new code injection (at 2 minutes 45 seconds) (b) 7 faulty nodes are booted 60 seconds after new code injection (at 3 minutes).


