Towards Efficient Stabilizing Code Dissemination in Wireless Sensor Networks

Sain Saginbekov and Arshad Jhumka

Department of Computer Science
University of Warwick
Coventry, CV4 7AL, UK
Email: {sain,arshad}@dcs.warwick.ac.uk

One important component of network reprogramming is code dissemination, when the updated program code is distributed to the relevant nodes. Very few code dissemination protocols tolerate transient faults that corrupt the state and these faults can cause the old code to disseminate in the network. We propose two protocols called *BestEffort-Repair* and *Consistent-Repair* that transform fault-intolerant code dissemination protocols into non-masking fault-tolerant protocols where, eventually, all nodes obtain the new code. We conduct experiments with both protocols on TelosB-like motes and over TOSSIM simulations to show their correctness and also their performance. We conduct a case study whereby both protocols are added to a state-of-the-art code dissemination protocol, viz. *Varuna* to evaluate their impact on Varuna. Our results show that (i) Varuna, which is fault-intolerant, is transformed into a stabilizing code dissemination protocol (ii) they induce low overhead on Varuna, and causes all nodes to eventually receive the new code. BestEffort-Repair is biased towards fast recovery whereas Consistent-Repair attempts to reduce the number of erroneous downloads in the network. Our main contribution is the first corrector protocols that correct code dissemination in the presence of transient faults.

Keywords: Code dissemination; Transient faults; Non-masking fault tolerance; Wireless sensor networks; Error Detection

Received July 16, 2013

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. 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 or treacherous 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) code 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 code dissemination aspect of network reprogramming.

Several code 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 tolerates transient data faults, i.e., data faults that corrupt the state of the code dissemination protocol. Transient data faults, which are also known as *soft errors*, are known to occur in WSNs [6, 7, 8, 9]. Given that several code dissemination protocols work by advertising the metadata, viz. version number[1], of the new code, e.g., [3, 4, 10, 11], any corruption of the version number in the advertisement messages or those stored at the nodes can, in the worst case, lead to the network nodes having stale code, thereby reducing the ability of the network to perform properly. Similarly, any state corruption can lead, in the worst case, to nodes downloading stale code (we will show this later). Thus, it is important to make these code dissemination

[1]In this paper, whenever we say metadata, we mean version number.

protocols tolerate these transient data faults.

When a node decides that another node has an updated code fragment, generally, it makes a request to the updated node which, subsequently, sends (i.e., broadcasts) the update to the requesting node. The process of advertising code updates, sending code requests and downloads is energy consuming. As the communication part consumes a large portion of energy and the significant amount of energy per transmitted bit used [12], and given the size of codes, which may vary from 20 bytes up to tens of kilobytes [2, 10, 13, 14, 15, 16], code dissemination consumes a significant amount of energy. This is exacerbated when transient faults occur, as nodes may mistakenly request and download stale code. In the worst case, the whole network may download the stale code, at great energy expense.

There exists a hierarchy of fault tolerance properties namely fail-safe fault tolerance (which ensures that a program always satisfies its safety specification), non-masking fault tolerance (which ensures that a program always satisfy its liveness specification) and masking fault tolerance (which ensures that both safety and liveness are satisfied, even in the presence of transient faults) [17]. In the context of code dissemination, fail-safe fault tolerance amounts to a node not downloading any code if it believes the code to be old. Masking fault tolerance means that all nodes will only download the new code (as if no fault has occurred), and only once. On the other hand, non-masking fault tolerance allows some erroneous downloads (i.e., downloads of old code) before eventually all nodes download the updated code. Such erroneous downloads are only allowed to occur finitely, though.

Given that code update dissemination is energy consuming, it is thus preferable to reduce the number of erroneous downloads while ensuring that all nodes eventually download the updated code. Fail-safe fault tolerance is not suitable as it means that some nodes may not update (which will impact of the usefulness of the network). Masking fault-tolerant code dissemination protocols would minimise the number of erroneous downloads but, given the nature of the WSNs and of the dissemination process, masking fault tolerance is not practical. On the other hand, non-masking fault tolerance means that nodes may erroneously download old code only finitely, but they will eventually download the updated code. However, a small number of erroneous downloads can be tolerated if this means that the network state is consistent, allowing the proper dissemination of the updated code.

To design non-masking fault tolerance, it is both necessary and sufficient for a program to contain a specific type of fault tolerance component called a *corrector*. A corrector is a class of program component that enforces a predicate on the execution of a program. There exists different correctors that can guarantee non masking fault tolerance for a given program, however they may differ in their efficiency. In this work, instead of proposing a specific non-masking fault-tolerant code dissemination protocol[2] for WSNs, we address the problem in a different way: we first provide an abstract specification of the code dissemination problem, and based on the definition, we propose (i) a definition of a corrector protocol and (ii) two *corrector* protocols, called *BestEffort-Repair* and *Consistent-Repair*. Each can be added to any existing (fault-intolerant) code dissemination protocol to transform it into a fault-tolerant code dissemination protocol. Specifically, since the corrector protocol is designed based on the code dissemination specification, rather than on an actual implementation, if the corrector is added to any code dissemination implementation that satisfies the dissemination specification, then the resulting protocol is non-masking fault tolerant [18]. Further, to detect state corruption, a detector component, which detects the validity of a predicate in a given state, is designed based on the protocol implementation.

The two corrector protocols developed has enabled us to observe a tradeoff during recovery: Consistent-Repair results in a lesser number of erroneous downloads than BestEffort-Repair. However, BestEffort-Repair has a shorter completion time in that Consistent-Repair needs more time to make better update decision. A shorter recovery time means that the network state becomes consistent faster, and can perform useful work faster.

Overall, our approach is as follows: *given a fault-intolerant code dissemination protocol, we design a protocol-specific detector together with a generic corrector component to obtain a corresponding non-masking fault-tolerant code dissemination protocol.*

Contributions In this context, we make the following contributions:

- We formalise the concept of code dissemination in WSN, and provide three refined specifications, viz., strong, consistent and best effort code dissemination.
- We show that (i) there is no deterministic algorithm that solve strong code dissemination in presence of transient faults, and (ii) there is no deterministic 1-local algorithm that solves strong code dissemination in presence of a stronger class of transient faults, called *detectable* faults.
- We present two novel f-local algorithms called (i) BestEffort-Repair and (ii) Consistent-Repair that, when added to any fault-intolerant code dissemination protocol, solves (i) BestEffort code dissemination and (ii) Consistent code dissemination, and we prove the correctness of both protocols.
- We run experiments and simulations on our protocol using TelosB platform-based motes and TOSSIM [19], respectively, and show their correctness and performance, especially the locality

[2]Henceforth, whenever we refer to fault tolerance in code dissemination, we mean non-masking fault tolerance.

property of the protocols.
- We present a case study where we add both protocols to an existing code dissemination algorithm, namely Varuna [4]. We instrument Varuna with a specific detector which triggers the protocols upon detection of an error. We show that both BestEffort-Repair and Consistent-Repair induce very little overhead on Varuna in presence of detectable 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 both protocols, 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 Section 3, we present the system and fault models assumed in the paper. We present a definition and specifications for code dissemination in the context of network reprogramming in Section 4. We present some theoretical results in Section 5. In Section 6, we present two f-local corrector algorithms that stabilise the code dissemination of code updates. We present the experimental setup and results to evaluate the performance of the proposed algorithms in Section 7. In Section 8, we present a case study where we add the algorithms to an existing code dissemination protocol to show the viability of our approach. We discuss aspects of our approach in Section 9, and we conclude the paper in Section 10.

2. RELATED WORK

2.1. Code Dissemination

There currently exists several dissemination protocols which update the running code on nodes to new ones. While some of the protocols deliver complete binary image of the code, like [2, 10, 13, 20], some other protocols deliver only the difference between the new code and the old code [21, 22]. There also exists protocols which deliver tasks [23], network parameters [24], and queries [25].

In XNP [13], 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 [20] 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 [10] allows large data transmission by fragmenting data into fixed-size pages. It also supports pipelined page transmission to make dissemination faster. Unlike MOAP, the nodes in Deluge should not wait for complete code image before forwarding it. The authors of MNP[2] propose another protocol like Deluge which fragments the code image and uses the 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 addresses the hidden terminal problem. Also, in MNP, some of the nodes can go to the sleep mode to save energy whenever there is no data to receive or transmit.

Because of the features of WSNs, such as transient link failures and node mobility, not all nodes may update their code during dissemination phase. The Trickle algorithm [3] addresses this problem by using a "polite gossip" policy. In Trickle, every node broadcasts advertisement messages about the code. The advertisement message is basically metadata, that includes version number[3], 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 τ up to τ_h, which is upper bound for τ. If it hears different metadata τ becomes τ_l, which is lower bound for τ.

Varuna [4] is another protocol which supports code update maintenance. This protocol saves energy in the steady phase,where no dissemination is being done. Unlike Trickle, where there is a linear increase of energy consumption - due to polite gossiping, energy consumption in Varuna is constant in steady phase (i.e., when there is no new code update in the network). To achieve constant energy consumption in steady phase, nodes in Varuna send advertisement messages only when there is a change in the neighbourhood topology or metadata since its last advertisement transmission. Finally, there exists code dissemination frameworks that target the reconfigurations of a subset of nodes in the network, rather than requiring all the nodes to be updated [26, 27].

These protocols do not consider transient memory faults that corrupt the state of the running code dissemination program, including the protocol messages, which may lead to the dissemination 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 code dissemination in the presence of transient faults.

2.2. Fault Tolerance

In [28], it has been shown that a class of components, known as correctors, is sufficient to design non-masking fault tolerance. 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 stabilising fault-tolerant one. Correctors are components that enforce a given predicate on program executions, whenever the predicate has been violated. The area of

[3]Henceforth, we will use the terms metadata and version number interchangeably.

self-stabilization is mature, and several stabilizing algorithms exist for several important problems [29].

2.3. Reliable Broadcast

Since we focus on code dissemination protocols, the problem of reliable broadcast is relevant. We provide a brief survey here.

The work proposed in [30] is one of the earliest work that deal with the broadcast problem in multihop radio networks. They propose fault-tolerant broadcasting algorithms and give their asymptotic bounds on completion time. They assume that faulty nodes are permanent of unknown locations and do not receive and send messages. Also they consider line and grid as an underlying network topology.

In [31], the author shows that it is possible to obtain reliable broadcast whenever the number of Byzantine nodes, nodes which may behave arbitrarily, f, is no more than some value. And this f is defined in terms of a communcation range r. Moreover, the author shows that it is impossibe to obtain reliable broadcast when f is bigger than some threshold value. The work assumes a grid network and the existence of a prefixed schedule and everyone follows to this schedule to avoid collisions. In [32], the authors improve on [31] by making possibility bounds tighter. In particular, it has been shown that it is possible to achieve reliable broadcast when the number of faulty(Byzantine) nodes is strictly less than the threshold value for which in [31] it is showed that it is impossible to achieve reliable broadcast.

In [33], unlike the previous work where there is no address spoofing and collision, the authors relaxed this assumption and showed that reliable broadcast is possible even in the presence collisions and address spoofing as long as they are bounded and the number of faulty nodes is less than some threshold value.

In [34], the authors address the broadcast problem in the presence of Byzantine faults with faulty nodes having bounded number of messages m_f. They show the possibility of reliable broadcast whenever the number of messages, m, of the correct node is lower bounded by some value defined in terms of m_f. They assume the existence of a prefixed time-slotted schedule, but faulty nodes may not follow the schedule thereby making collisions.

In [35], the authors propose a protocol which is *safe*, i.e., correct nodes do not download an incorrect message. The protocol guarantees this property whenever $D \geq H + 2$, where D is the shortst distance between two Byzantine nodes and H is a prorocol parameter which is assumed to be known by all correct nodes. The paper also discusses the possibility of reliable broadcast in the *torus* network whenever $D \geq 5$ and $H = 2$. The same authors generalized this result to planar graphs in [36]. In particular, the authors show that for $D > Z$, where Z is the maximal number of edges per polygon, it is possible to achieve reliable broadcast.

3. MODELS: SYSTEM AND FAULTS

3.1. Graphs and Networks

We define a wireless sensor node as a computing device equipped with a wireless interface and associated with a unique identifier. Communication in wireless networks is typically modelled with a circular communication range centred on the node. With 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 $G = (V, A)$, where V is the set of wireless sensor nodes of size $|V|$, and A is a set of arcs or directed links. Each directed link is an ordered pair of distinct nodes (m, n), meaning node m can communicate with node n. For a directed link (m, n), we call n (respectively, m) a *downstream neighbour* (respectively, a *upstream neighbour*) of m (respectively, n). We denote by M_d (respectively, M_u), the set of m's downstream (respectively, upstream) neighbours. We also assume that, for every node m, $M_d, M_u \neq \emptyset$. Whenever we say a node n sends (resp. receives) a message, we mean n sends (resp. receives) the message to its downstream (resp. from its upstream) neighbours

The d-hop neighbourhood of a node m, denoted by M^d, is a set of nodes such that the length of the shortest path from m to a node in the set is at most d. We say that two nodes m and n can collide at node p if $(m, p), (n, p) \in A$[4].

3.2. Distributed Programs

We model the processing on a WSN node as a process containing non-empty sets of variables and actions. A distributed program P is then a finite set of communicating processes. We represent the communication network of a distributed program by a directed connected graph $G = (V, A)$, where V is the set of processes and A is a set of directed links. A link $(m, n) \in A$ means that a process m can communicate with a process n.

A variable v_i takes values from a fixed and finite domain D_i. We denote a variable v_i of process n by $n.v_i$. Each process n has a special channel variable, denoted by $n.ch$, modelling a FIFO queue of incoming data sent by other nodes. This variable is defined over the set of (possibly infinite) message sequences. Every variable of every process, including the channel variable, has a set of initial values. The *state* of a program P is an assignment to variables of values from their respective domains. The set of *initial states* is the set of all possible assignments of initial values to variables

[4]We will say two nodes m and n can collide if such a node p exists.

of the program. A state is called *initial* if it is in the set of initial states. The *state space* of the program is the set of all possible value assignments to variables. An action a at process n updates one or more variables of n atomically.

3.3. Semantics

3.3.1. Program

We model a distributed program as a transition system $P = (\Sigma, I, \Delta)$, where Σ is the state space, $I \subseteq \Sigma$ the set of initial states, and $\Delta \subseteq \Sigma \times \Sigma$ the set of state transitions (or steps). A computation of P is a maximal sequence of states $s_0 \cdot s_1 \ldots$ such that $\forall i > 0, (s_{i-1}, s_i) \in \Delta$. If the computation is finite, then it terminates in a *final* state. A state s of a computation is final if there is no state s' such that $(s, s') \in \Delta$.

In a given state s, several processes may be ready to execute, and a decision is needed to decide which one(s) execute. A *scheduler* is a predicate over the set of computations. In any computation, each step (s, s') is obtained by the fact that a non-empty subset of enabled processes atomically execute an action. This subset is chosen according to the scheduler. A scheduler is said to be *central* [37] if it chooses only one ready process to execute an action in any step. A scheduler is said *distributed* [38] if it chooses at least one ready process to execute an action in any execution step. A scheduler may also have some fairness properties [29]. A scheduler is *strongly fair* if every process that is ready infinitely often is chosen infinitely often to execute an action in a step. A scheduler is *weakly fair* if every continuously ready process is eventually chosen to execute an action in a step. A *synchronous* scheduler is a distributed scheduler where *all* ready processes are chosen to execute an action in a step.

In this paper, we assume a synchronous scheduler, capturing a synchronous system where an upper bound exists on the time for a process to execute an action. This assumption is not unreasonable as WSNs are often time-synchronized to either correlate sensor readings from different devices. Overall, in this paper, we assume a *synchronous system model*.

3.3.2. Specification

A specification is a set of computations. A program P satisfies a specification Φ if every computation of P is in Φ. Alpern and Schneider [39] stated that every computation-based specification can be described as the conjunction of a safety and liveness property. Intuitively, a safety specification states that something bad should not happen, i.e., the safety specification defines a set of computation prefixes that should not appear in any computation. On the other hand, a liveness specification states that something good will eventually happen, i.e., the liveness specification specifies a set of state sequences such that every computation has a suffix in the set.

We assume the specification to be fusion-closed and suffix-closed. A specification is fusion-closed if two computations $\alpha \cdot s \cdot \beta$ and $\lambda \cdot s \cdot \gamma$ are allowed by the specification, then so are the computations $\alpha \cdot s \cdot \gamma$ and $\lambda \cdot s \cdot \beta$. A specification is suffix-closed if, for every computation allowed by the specification, then so are the suffixes of the computation. The assumption of fusion closure is, in general, reasonable given that conventional specification and implementation languages are fusion closed. Further, any non-fusion closed specification can be transformed into an equivalent fusion closed one, through the addition of history information [40].

3.3.3. Communication

We model synchronous communication as follows: after a process i broadcasts a message in state s_i, all downstream neighbour processes execute the corresponding receive in state s_{i+1}, i.e., the corresponding receive is executed before any other enabled actions of process n, such that, in some sense, message deliveries take higher priority.

3.4. Faults

Faults typically occur in wireless sensor networks. Due to limited resources such as computing, memory and energy, harsh environmental conditions and buggy programs, wireless sensors may experience a number of different types of faults. As mentioned in [41], these faults can be classified as node failures and hardware faults, communication faults, and software faults. Some of these faults lead to transient memory corruptions [41]. There have been several works done that are tolerant to transient memory faults or present memory protection mechanisms from some actions that lead to this type of faults [42, 43, 44, 45]. Recently, as reported in [9], transient faults have occurred with a probability of approximately 0.1% in a large scale deployment and such transient faults severely impact on the efficiency of the protocols. Also, there exists a tool that is designed specifically for wireless sensors to emulate memory faults to check the reactions of software to these faults [46].

A fault model stipulates the way programs may fail. We consider *transient data faults* that corrupt the state of the code dissemination program by artificially corrupting the values held by the variables and messages. These faults are also known as soft errors. Formally, our fault model is a set F of faulty actions [28]. These are similar to program actions, as they may modify the variables of programs and thus alter the program state. We say that a fault occurs if a fault action is executed. Fault actions can interleave program actions and they might or might not be executed when enabled. We say a computation is F-affected if the computation contains program transitions and transitions from fault model F. We also

assume that the sink is able to retrieve an uncorrupted version of the code and version number (with the sink acting as a gateway), though the sink itself can be corrupted.

Definition 3.1 (Consistent State in Code Dissemination). *Given a network $G = (V, A)$ and a code dissemination program Ψ for G, the state of a neighbourhood G' of G is said to be* consistent in *s if there is at most 2 distinct version numbers in that neighbourhood in s, where s is a state of Ψ. A process is said to have a* consistent state in *s if its code's version number is the same as that of at least one other neighbour process in s. A state of Ψ is* consistent in *s if all neighbourhoods of G are consistent in s.*

Definition 3.2 (F-affected node and F-affected area). *Given a network $G = (V, A)$, a fault F that corrupts the program state, an area $G' = (V', A')$, with G' being a subgraph of G, is F-affected in a state s iff $\exists V'' \subseteq V'.\forall n \in V''$, n may need to change its state to make the state of V' consistent. We call such a node n an F-affected node in s*[5].

When a node changes its state to make the program state consistent, we say that the node *corrects its state.*

Definition 3.3 (Stabilizing algorithm). *Given a network $G = (V, A)$, a problem specification Φ for G, and an algorithm Ψ. Algorithm Ψ is said to be* stabilizing *to Φ iff every computation of Ψ has a suffix which is a suffix of a computation of Φ that starts in an initial state.*

Definition 3.4 (d-local stabilizing algorithm). *Given a network $G = (V, A)$, a problem specification Φ for G, and a stabilizing algorithm Ψ to Φ. Algorithm Ψ is said to be d-*local stabilizing *to Φ iff the cost of correcting the state of a node is bounded by functions of d.*

4. SPECIFICATIONS

In this section, we formally define the problem of code dissemination for network reprogramming. We then provide two refined specifications for solving the code dissemination problem: (i) deterministic code dissemination, and (ii) stabilising code dissemination.

4.1. Abstract Specification of Code Dissemination

Before we provide problem definitions, we introduce some notations we use in the rest of the paper. We denote a code fragment by (π, v_π), with π being the code and v_π being the version number of the code. We denote by v'_π a possibly corrupted version number for code π, i.e., if $v'_\pi = v_\pi$, then the version number is not corrupted, corrupted otherwise. We say that a code has code fragment (π^n, v'_{π^n}) to mean that a node n has the code fragment (π, v_π) but has version number v'_π associated with it instead. Thus, unless stated otherwise, whenever we say a node n has code fragment π, we mean a node n has code fragment (π, v'_π).

We assume the version number to be a scalar quantity. We also assume that the version number can grow arbitrarily large.

Definition 4.1 (Code Update). *Given two code fragments (π, v_π) and (Π, v_Π), Π is said to be an* updated code over *π if $v_\Pi > v_\pi$. If a node n_i changes its code from π to Π and Π is a updated code over π, then we say that a* node *n_i updates* its code to Π. *Otherwise, if a node n_i changes its code from Π to π, then we say that a* node *n_i outdates* its code to π.

Definition 4.2 (New Code for G). *Given a code fragment (Π, v_Π) and a network G, we say that Π* is a new code for *G if all nodes in G have code fragment π and $v_\Pi > v_\pi$.*

We will say that Π is an updated code to mean that Π is an updated over code π, whenever π is clear from the context. We will also say that a node n updates/outdates its code to Π/π if Π and π is obvious from the context.

We now provide an abstract definition of code dissemination as part of a network reprogramming protocol.

Definition 4.3 (Code Dissemination (CD)). *Given a network $G = (V, A)$, with a dedicated node called a sink $S \in V$, and an updated code (Π, v_Π) to be disseminated. Then, a code dissemination for Π is a sequence of sets of receivers $\langle R_0 \cdot R_1 \dots R_{k_\Pi} \rangle$ such that*

1. $R_0^\Pi = \{S\}$
2. $\forall\, i, 0 \leq i \leq (k_\Pi - 1) : \forall r \in R_{i+1}^\Pi, \exists s \in R_i^\Pi \cdot (s, r) \in A$
3. $\bigcup_{0 \leq i \leq k_\Pi} R_i^\Pi = V$

Given a network G and an updated code Π, the code dissemination process starts with the sink (condition 1). Then, the code update process propagates forward (condition 2), one hop at a time, until all the nodes have received the updated code (condition 3). The sequence represents the sequence in which the nodes updates their code.

In the above definition, we have made three assumptions: (i) when a code dissemination process starts, all the nodes have the same code base, i.e., they all have the same code, (ii) all nodes need to get the updated code (however, we can easily adapt the definition to the case where only a subset of nodes require the code update), and (iii) only one code dissemination can take place at a time, i.e., a code update can only occur once a previous one has completed. We call k_Π, the *dissemination latency* for Π. Observe that condition 2 *does not warrant that all the downstream neighbours* of an updated node to

[5] We will only say F-affected if the state s is obvious from the context.

receive the code update in the next round. Due to issues such as message collisions and duty cycling, a downstream neighbour node may not receive the update in the next round, but sometime later from, possibly, another upstream neighbour.

4.2. Local Specifications for Code Dissemination

The specification given in 4.4 is a global specification in the sense that it specifies the expected behaviour at the network level. In a distributed system, the verification that a program satisfies the global specification is challenging, given that global state is not instantaneously available. Thus, it is preferable to develop node-level specifications, which we call *local specifications*, which are more amenable to verification. The combination of local specifications (one for each process) result in the global specification.

We now present three increasingly weaker local specifications through which code dissemination could be achieved, which we call (i) strong code dissemination (CD), (ii) consistent CD and (iii) best effort CD. The first specification, strong CD, represents a gold standard and is satisfied by current dissemination protocols, such as [2, 3, 4]. The weaker specifications become important especially when transient faults occur in the network. We will define both specifications in terms of safety and liveness [39].

4.2.1. Strong Code Dissemination

Intuition In the fault-free case, a node n, having code π, will only download a code from a neighbor node, having code Π, if Π is an updated code. Further, n will not download Π again. Current code dissemination protocols also guarantee that, eventually, every node will download the updated code (even if some nodes are temporarily disconnected from the network, due to duty-cycling, link failures etc).

Thus, we define strong CD as follows (Definition 4.5):

Definition 4.4 (Strong CD). *Given a network $G = (V, A)$, a node $n \in V$ having a code fragment π, and a new code fragment (Π, v_Π) for G. Then,*

- Accuracy*: Node n will only change its code to an updated one.*
- Update*: Eventually node n will permanently update its code to Π.*

The liveness part of the specification, i.e., the update property, for strong code dissemination ensures that $\bigcup_{0 \leq i \leq k_\Pi} R_i^\Pi = V$ (see Definition 4.4). On the other hand, Definition 4.5, through the accuracy property, puts an additional constraint on code dissemination in that nodes only change their code with an updated one. In other words, $\bigcap_{0 \leq i \leq k_\Pi} R_i^\Pi = \emptyset$, i.e., no node updates more than once. In general, in the absence of faults, it can be expected that the system will satisfy the strong CD specification, e.g., [3, 4]. However, due to external factors, such as transient faults, nodes may wrongly outdate their codes and, if they do so, they will eventually have to correct these mistakes.

These wrong code changes (i.e., code outdates) give rise to various possible weaker specifications, namely (i) consistent CD and (ii) best effort CD.

4.2.2. Consistent Code Dissemination

Intuition When transients fault occur, it may be the case that a node n cannot distinguish between updated and outdated code. Specifically, a node n that has already updated may download the old code from a neighbour, i.e., n becomes outdated, if it believes that the neighbour has the code update. This is not an ideal situation. However, a node m that has yet to be updated may believe that another node m', with the same code as m, has the code update and may wrongly download the same (old) code. This specification forbids a node to outdates itself.

The specification

Definition 4.5 (Consistent CD). *Given a network $G = (V, A)$, a node $n \in V$ having a code fragment π, and a new code fragment (Π, v_Π) for G. Then,*

- No outdate*: Node n will never change its code to an outdated one.*
- Update*: Eventually node n will permanently update its code to Π.*

4.2.3. Best Effort Code Dissemination

Intuition The best effort CD specification allows for an updated node n to outdate itself as n may wrongly believe some node m to have the new code while n has the old code.

Definition 4.6 (Best Effort CD). *Given a network $G = (V, A)$, a node $n \in V$ having a code fragment π, and new code fragment (Π, v_Π) for G. Then,*

- Eventual accuracy*: Eventually, node n will only change its code to an updated one..*
- Liveness*: Eventually node n will permanently update its code from π to Π.*

An example is used to help better understand and differentiate between the three different specifications proposed. At the start of the dissemination of the new code Π, the old code π resides at every node in the network. Assume that the old code has version 0 and the new code has version 1. Hence, for any node n in the network, there are four possible code transitions in the presence of transient faults:

1. $0 \rightarrow 0$: Node n has the old code and changes to the old code again - Redundant
2. $0 \rightarrow 1$: Node n has the old code and changes to the new code - **Code update**
3. $1 \rightarrow 0$: Node n has the new code and changes to the old code - **Code outdate**

4. $1 \rightarrow 1$: Node n has the new code and changes to the new code again - Redundant

The strong CD specification only allows the second type of code transition. The consistent CD specification allows the 1, 2 and 4 types of code transitions while the best effort CD specification allows all of them. It can be observed then that BestEffort specification allows more redundant downloads than either of the other two specifications.

4.3. Fault Tolerance Issues: An Overview

It can happen that a code dissemination protocol that has been proved correct (i.e., satisfies its strong specification in the absence of faults), violates its specification in the presence of faults due to it not being able to handle faults [28], i.e., the code dissemination protocol is fault-intolerant. As such, there is a variety of fault tolerance properties that the program can satisfy, viz. fail-safe fault tolerance, non-masking fault tolerance and masking fault tolerance [28]. A fail-safe fault-tolerant program guarantees that safety will always be satisfied, while a non-masking fault-tolerant program guarantees that liveness will eventually be satisfied, even if safety can be temporarily violated. On the other hand, a masking fault-tolerant program (the gold standard) guarantees that the program will satisfy its specification even in the presence of faults. To transform a fault-intolerant program into a fail-safe fault-tolerant (resp. non-masking fault-tolerant) program, addition of program components called detectors (resp. correctors) to the fault-intolerant program are both necessary and sufficient. Thus, to make a program masking fault-tolerant, it is necessary and sufficient to add both detectors and correctors [28]. A detector component is one that asserts the validity of a predicate in a running program, while a corrector component enforces a predicate on a running program.

In networking, a non-masking fault-tolerant program is generally suitable as it guarantees that, eventually (i.e., when faults stop), the program will satisfy its specification again. Though non-masking fault tolerance entails the erroneous downloads of codes, it is the one more suited to code dissemination as masking fault tolerance is very expensive, both spatially and temporally, to guarantee. Given the existence of several code dissemination algorithms, it is not intended, in this work, to develop another (non-masking) fault-tolerant code dissemination protocol for network reprogramming. The thrust is to *develop a generic corrector protocol* that, when added to a fault-intolerant code dissemination protocol, will enable the resulting protocol to satisfy the liveness specification (i.e., eventually all nodes will permanently update with the updated code). We also require that the corrector protocol is only executed when an erroneous state is detected.

At this point, we need to define the properties of such a corrector protocol that will capture its correctness. Since we wish this corrector protocol to be generic, and work as a wrapper (i.e., it can plug in with various code dissemination protocols), it cannot be based on any specific code dissemination protocol implementation. Rather, the working of the corrector protocol should only be based on the specification of the code dissemination protocol, more specifically its interface and specification. Such an approach is what has been termed as graybox stabilization [18].

Definition 4.7 (Corrector Component for Code Dissemination). *Given a strong code dissemination specification σ^s (Definition 4.5) and a weaker version σ^w (Definitions 4.6 or 4.7), some transient fault model F, a protocol Σ that satisfies σ^s in the absence of F but violates σ^s in the presence of F, and a program ϕ. Then, ϕ is a* σ^w-corrector program *for σ^s iff*

- Transparency: *In the absence of F, $(\Sigma \circ \phi)$ satisfies σ^s.*
- Stabilizing: *In the presence of F, $(\Sigma \circ \phi)$ satisfies σ^w.*

If σ^w is consistent CD (resp. best effort CD), then ϕ is a consistent (resp. best effort) corrector for strong CD.

Here, $A \circ B$ represents the addition of program A with program B [18]. The set of computations of the composite system $(A \circ B)$ is the smallest fusion-closed set that contains computations of A and B, with the initial states being the set of common initial states of A and B. Definition 4.8 stipulates that, when there is no transient fault in the network, the corrector program is transparent, i.e., it does not interfere with the working of the code dissemination protocol and satisfies the strong code dissemination. However, when transient faults are occurring, then the corrector program will help the code dissemination protocol to eventually guarantee that a node will permanently download the updated code, after possibly having downloaded stale code.

5. THEORETICAL RESULTS

In this section, we show that (i) it is impossible to solve the strong code dissemination problem in presence of transient faults, and (ii) there exists no 1-local algorithm to solve the strong code dissemination problem in the presence of a stronger class of transient faults, which we term as *detectable faults*.

5.1. Strong Code Dissemination in the Presence of Transient Faults

In this section, we investigate the possibility of developing an algorithm that solves the strong code dissemination problem in the presence of transient faults. Ideally, even in the presence of transient faults, it would be beneficial if a node only updates with new

code to prevent redundant downloads, thereby saving energy.

Intuition From the specification of deterministic code dissemination (Definition 4.5), it is stated that nodes only update their codes when they are in the presence of a newer code fragment. However, when transient faults occur, the version number that is advertised by or stored at a node can be corrupted, possibly leading to nodes downloading old code fragments. Thus, the first main contribution of the paper is captured by Theorem 5.1, which states that it is impossible to solve the strong code dissemination in the presence of transient faults.

Theorem 5.1 (Impossibility of strong CD). *Given a network $G = (V, A)$, a fault model F that corrupts the program state, and an updated code fragment (Π, v_Π). Then, there exists no deterministic algorithm that solves the strong code dissemination problem for Π in G in the presence of F.*

Proof. Consider the network G, and assume a deterministic algorithm Ψ that solves the strong code dissemination problem. We will construct an appropriate state and show that, under Ψ, a node may wrongly update, hence a contradiction.

Assumptions: Two nodes n_i and n_j where n_i (resp. n_j) is both an upstream and downstream neighbour of n_j (resp. n_i).

Consider a fault free computation $C = s_0 \cdot s_1 \ldots$ of Ψ. In a given state s_k in C, two nodes $n_i, n_j \in V \setminus \{S\}$ have the following code fragments: n_i has Π and n_j has π.

Now, nodes n_i and n_j interact such that n_i and n_j inform each other of their respective code fragments, i.e., about their respective version numbers. Given that Ψ solves the strong code dissemination problem, node n_j will eventually permanently update its code with Π in a state $s_l, l > k$ as Π is an updated code fragment.

Now, consider a faulty computation $C' = s'_0 \cdot s'_1 \ldots$ of Ψ, and a state s'_k which is exactly the same as s_k (above) except for the following: (i) node n_i has a code fragment $(\Pi_{n_i}, v'_{\Pi_{n_i}})$ and node n_j has code fragment $(\pi_{n_j}, v'_{\pi_{n_j}})$. In a state $s'_l, l > k$ of C', assume that n_i has the same version view as n_j in s_l and n_j has the same version view as n_i in s_l.

Since Ψ is deterministic and solves strong CD, node n_i will permanently update its code with π in s'_l, which is a contradiction as π is an old code. Hence, no such deterministic Ψ exists. □

The impossibility is underpinned by some major problems, the most prominent being: (i) Nodes are not able to detect unexpected version numbers as, for example, if nodes with the stale code have their respective version numbers corrupted to very high values, old code may propagate through out the network, and (ii) nodes with the updated code may have their respective version numbers corrupted to that of the old code, while nodes with the old code have their respective version numbers corrupted to the new one, i.e., all updated nodes appear as outdated and all outdated nodes appear as updated.

To attempt to circumvent this impossibility of Theorem 5.1, there are different possible avenues. For example, one may allow algorithms to make a finite number of mistakes, thereby solving a weaker problem specification (such as the weak code dissemination). Another example might be to solve the strong code dissemination problem in presence of a *stronger* fault model, i.e., a fault model where the set of possible corruptions is constrained. For the first possibility, if the possibly few updated nodes are overwritten with the old code, then there is no chance of the network getting the new code. Thus, even if a finite number of mistakes are allowed (i.e., download of old code), then there is no guarantee of the network getting the correct code. Thus, in this paper, we follow the second possibility, i.e., we assume a stronger fault model.

Thus, the assumed fault model needs to be such that the two stated problems are handled: *we require the faults to result in detectable errors.* Thus, we rule out a few fault actions: (i) we require that the fault model does neither make outdated nodes appear as updated and updated as outdated, (ii) nodes in a neighbourhood need to be corrupted differently (so nodes in a neighbourhood do not appear as outdated/updated). We call the resulting fault model as the *detectable* fault model, which we assume in the rest of the paper. An example of a possible fault ruled out by the first constraint can be illustrated with an example: assume the old code version is 1 and the new version is 2, and the version number increases by 1 for each new update. An updated (resp. outdated) node cannot have its version number to be corrupted to 1 (resp. 2). Also, say a node n has its version number corrupted to 5, then a node m in n's neighbourhood cannot have its version number corrupted to 6, as m will appear as updated to n.

In the presence of detectable faults, a trivial solution to solve the strong code dissemination is to require the sink to periodically start the dissemination process. Whenever a node encounters a version that is unexpected (allowed under the detectable fault model), it does not need to download the code associated with it. When it sees a version number that is expected (i.e., realistic), and since the code associated with the version number cannot be a stale one (as it is ruled out by the fault model), the node can download the code. Unfortunately, such a scheme is expensive as the network will need to spend lots of energy for dissemination, i.e., the protocol is a global one. Thus, we seek to determine whether nodes can rely only on its 1-hop neighbourhood for code dissemination (just as in a fault-free case) in the presence of detectable faults. This is captured in Theorem 5.2.

Theorem 5.2 (Impossibility of 1-local strong CD). *Given a network $G = (V, A)$, a detectable fault model F, and an updated code fragment Π. Then, there exists no 1-local algorithm that solves the strong code dissemination problem for Π in G in the presence of F.*

Proof. The proof is trivial. If the 1-hop neighbourhood of a node is corrupted in such a way that the version numbers are either unexpected ones or old ones, then the node will not download any code. Hence, a 1-local protocol is not possible. □

Intuitively, if a neighbourhood is corrupted by a detectable fault model, then nodes will need to start downloading from uncorrupted nodes outside of the corrupted neighbourhood. Hence, this points towards a f-local algorithm, where f is the diameter of the affected area, that can solve the strong code dissemination algorithm. However, given the nature of WSNs and of the code dissemination process, strong code dissemination in the presence of faults is not appropriate, due to the overhead it induces on the network. To this end, we focus on the two other specifications, viz. best effort CD and consistent CD. In the next section, we present two corrector programs that, when added to a fault-intolerant code dissemination protocol, solve the BestEffort CD and Consistent CD problems.

6. CODE DISSEMINATION CORRECTION: TWO GENERIC CORRECTOR PROTOCOLS

In this section, we present two generic *corrector* programs, namely (i) *BestEffort-Repair* and (ii) *Consistent-Repair*. Each of the two protocols can be added to a fault-intolerant code dissemination protocol to make the code dissemination protocol satisfy some correctness specification.

As stated before, rather than developing a single (non-masking) fault-tolerant code dissemination protocol, the focus in on transforming existing fault-intolerant code dissemination protocols into non-masking fault-tolerant ones. To enable this, we adopt the technique for graybox stabilisation [18] whereby, rather than developing a corrector for a particular code dissemination protocol, a (generic) corrector protocol is designed based on a specification. This corrector can then be added to any implementation that satisfies the specification, resulting in the eventual program to be non-masking fault-tolerant. In that way, the corrector is reusable.

Further, from the definition of a corrector component (Definition 4.8), the corrector should be transparent to the code dissemination protocol when there are no faults is the network, i.e., the behaviour of the composite corrector and dissemination protocol should be identical to that of the dissemination protocol alone in the absence of faults. To achieve this, we include a detector component in the code dissemination protocol that, when satisfied (during faulty periods), triggers the corrector component. It would be advantageous to then be able to develop a detector based on a specification. However, in such a case, the efficiency of the detector is not very high, in that it can suffer from high false positives or false negatives, which can then cause the corrector to violate its transparency property. To compensate, we design the detectors based on protocol implementations, i.e., a detector is needed for each different code dissemination protocol. Overall, our approach is to develop a protocol-specific detector which, when its corresponding detection predicate becomes true, triggers the execution of the generic corrector program, making the code dissemination protocol non-masking fault-tolerant.

A design methodology suggested for graybox stabilisation is to design a program that contains two different components [18]: (i) a process-specific component and (ii) an interprocess-specific component. The process-specific component is responsible for making the state of a single process consistent, whereas an interprocess-specific component is responsible for correcting any inconsistency between different processes. In a fault-intolerant code dissemination protocol, since the only relevant information nodes keep about the code is the version number, then state inconsistency at the process level is irrelevant, i.e., the state of a single process is trivially consistent. On the other hand, state inconsistency can be detected when comparing the version numbers of two different processes. Thus, interprocess-specific component of a corrector program only needs to correct the states of processes that are inconsistent with each other.

In Sections 6.1 and 6.2, we present two corrector protocols that correct any state inconsistency between processes, transforming the fault-intolerant code dissemination protocols in non-masking fault-tolerant ones. The BestEffort-Repair protocol, as the name suggests, attempts to correct the state inconsistencies as fast as possible, while the Consistent-Repair protocol attempts to correct the state consistencies as intelligently as possible. In other words, the worst case scenario for BestEffort-Repair may be worse than that of the Consistent-Repair but the best case scenario for BestEffort-Repair is also better than that of Consistent-Repair.

6.1. The *BestEffort-Repair* Protocol

Before describing the *BestEffort-Repair* protocol and giving its formal description, we present the main idea behind it, and the special packets it uses. Since 1-local fault tolerance is not possible (Theorem 5.2), the main idea is to correct (i.e., repair) the protocol state as fast as possible. Correcting a state inconsistency (i.e., error) quickly means that the error does not propagate through out the whole network.

BestEffort-Repair uses six special types of data

packets (we call them BestEffort-Repair packets), which we describe below.

- **Prob**: It contains the code's version number and it is used to ask a neighbouring node to correct an error.
- **Check**: A node sends a Check packet to request the current version number of neighbouring nodes.
- **Rep**: A node sends a Rep packet in response to a Check packet and it contains the node's (stored) version number.
- **OK**: It is used to release some nodes from the correction process.
- **Cor**: A node sends a Cor packet to inform other neighbouring nodes about the correct version number..
- **Hello**: A node sends a Hello packet to inform other neighbouring nodes about the correct version number and also that it has the updated code.

Informally, BestEffort-Repair works as follows: When a node n_1 detects an error after communicating with a node n_2, it attempts to correct the erroneous state. A Prob packet is sent by n_1 to n_2 to indicate a problem, asking n_2 to correct the problem. If the error cannot be corrected by n_2, then Check packets are broadcast, creating a *correction tree*, rooted at the node (n_1) that detected the error. The leaf nodes of the tree responds to Check packets by sending Rep packets. If n_2 detects an error with any of the leaf nodes, it will spawn a subtree, within the main correction tree. Once a region in the network is reached where no fault has occurred, i.e., outside of the fault-affected area, then no more subtree is spawned. This means that a node's, say n_l, neighbourhood (i.e., all the children of the node within the correction tree) have the same code version, as the version is correct (under the detectable fault model). In other words, n_l has received Rep packets from its children with the same version number. Then, ultimately, the node n_l responds through a Hello or Cor packet, and its subtree "disappears". Any node sending a Hello or Cor packet will cause its subtree to "disappear" since the node has ascertained the correct version number (and in the case of Hello packet, n_l also notified its parent about the availability of the code as well).

6.1.1. *BestEffort-Repair: An Overview*

When a node n_1 detects an error (which is protocol-specific) after receiving a message from a neighbouring node n_2, n_1 sends a Prob packet to n_2, thereby asking n_2 to check whether it is the source of the error (we will shortly explain what happens if n_2 does not receive the Prob packet from n_1). Node n_1 then goes to the *Wait* state, where n_1 will wait for some predefined time. In turn, n_2 asks its neighbouring nodes, except n_1, for their version numbers by broadcasting a Check packet. Node n_2 then goes to the *Waitrep* state where it will wait for Rep packets from its neighbours over a certain time interval. All nodes that receive the Check packet from n_2 send a Rep packet to n_2. Now, node n_2 will compare all the received version numbers obtained from the Rep packets. If the version numbers are equal and match its own version number, then n_2 sends a Hello packet to n_1. By sending a Hello packet, node n_2 says to node n_1 that it has the correct version and it has the associated code too.

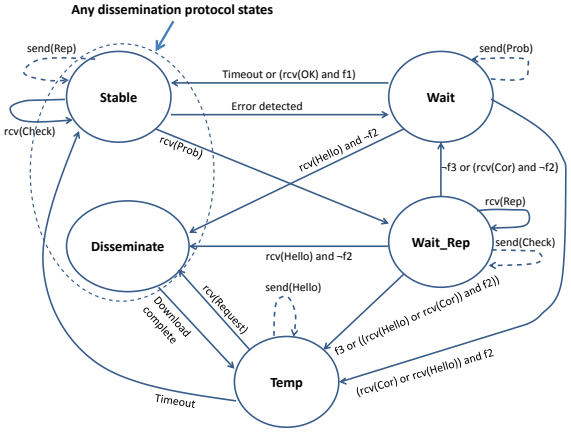


FIGURE 1. The state machine for BestEffort-Repair. Two states in dashed area are the states of any dissemination protocol. f1=TRUE if sender of *OK* packet is the node which sent *H*, f2=TRUE if Sender.Vers=Receiver.Vers, f3=TRUE if all received metadata are the same.

Now, if the received version numbers from the Rep packets are the same but differ from that of n_2, node n_2 will send a Cor packet to n_1 and corrects its code fragment by downloading from one of the Rep senders. By sending a Cor packet to n_1, node n_2 tells n_1 about the currently available version, which n_1 can download from another node with the associated code. If at least one of the received Rep packets contains a different version number i.e., the received version numbers are not identical, then n_2 goes to the *Wait* state and broadcasts a Prob packet, as done by n_1 earlier. This process continues until a node that sent a Prob packet will get a Hello or Cor packet. Nodes that were in the *Wait* or *Waitrep* state after updating their code fragments, go to the *Temp* state where they broadcast a Hello packet a few times.

Because of reasons such as transient link failures, a Prob packet sent by n_1 may not be delivered to n_2. To overcome this issue n_1 periodically sends a Prob packet to n_2 some predefined times until n_1 receives an implicit acknowledgement packet like Check, OK or Prob from n_2 or Hello or Cor from any node.

Figure 1 illustrates the state machine of BestEffort-Repair. The variables and (pseudo) code for the *BestEffort-Repair* algorithm is shown in Figures 2 and 3.

We now prove that BestEffort-Repair is a corrector component.

Variables of process i:

PacketType $\in \{H, Prob, Check, Hello, OK, Rep\}$
% H packets are application layer(data)/ code update maintenance packets.
% The other packets are Repair packets.

state $\in \{1, 2, 3, 4, 5\}$ **Init** state ==1;
//for PS, WAIT, WAITREP, TEMP, DISSEMINATE
% See Figure 1.

h, p, version, countH, countP : $\mathbb{N}$ **Init** countH:= 0, countP:=0
firstProb $\in \{0, 1\}$ **Init** firstProb == 1

TableProb, TableRep: {(id, version) : id $\in \mathbb{N}$, version $\in \mathbb{N}$}
% Keep track of nodes and version number they sent/receive

% Protocol Timers

PeriodProb: Timer
SendProb: Timer
Wait_Time: Timer
SendRep: Timer
WaitRep_Time: Timer
SendCheck: Timer
Temp_Time: Timer
SendHello: Timer
t: Timer
U: Timer % Parameter from application layer for periodic traffic.

FIGURE 2. Variables of BestEffort-Repair algorithm.

Lemma 6.1 (Containment of BestEffort-Repair). *Given a network $G = (V, A)$, detectable fault model F, an F-affected area $G' = (V', A')$, then, at most $O(|V'|)$ nodes will download the old code.*

Proof From BestEffort-Repair, a node n, after sending Check packets to its neighbours - due to receiving a Prob packet from a node n', waits for Rep packets. If the received Rep packets are all identical in their version numbers, then n will either broadcast a Cor packet (stating the expected correct version number and then download the code) or it will broadcast a Hello packet. If the version number is the old one, then, n will download the old code. All other nodes that receive the Hello packet will also download the old code. Thus, at most, all nodes in $m \in G'$ and all nodes $p \in M_u$ will receive a Hello packet with the version number being the old one. □

From Lemma 6.1, it can be observed that only a finite number of nodes, including updated ones, will change their code to the old one in presence of transient faults. Since there will then be the old code and the new code in the network, eventually, the code dissemination protocol will ensure that all nodes get the updated code. This is captured in Theorem 6.1

Theorem 6.1 (Correctness of BestEffort-Repair). *Given a network $G = (V, A)$, detectable fault model F, a strong code dissemination specification σ for G, BestEffort CD σ^b, a protocol Σ that satisfies σ in the absence of F but violates σ in the presence of F. Then, BestEffort-Repair is a* BestEffort-corrector component *for strong CD.*

Proof. For the transparency property, since BestEffort-Repair is only triggered when there is an error in the network, then safety is satisfied by the correctness of Σ. The stabilizing property follows from Lemma 6.1 and the nodes will the old code with ultimately download the correct code due to Σ. □

6.2. The *Consistent-Repair* Protocol

6.2.1. Consistent-Repair: An Overview
Consistent-Repair works in a similar way to BestEffort-Repair, with differences when a node changes its code.

When a node n_1 detects an error (which is protocol-specific) after receiving a message from a neighbouring node n_2, n_1 sends a Prob packet to n_2, thereby asking n_2 to check whether it is the source of the error (we will shortly explain what happens if n_2 does not receive the Prob packet from n_1). Node n_1 then goes to the Wait state, where n_1 will wait for some predefined time. In turn, n_2 asks its neighbouring nodes, except n_1, for their version numbers by broadcasting a Check packet. Node n_2 then goes to the Waitrep state, where it will wait for Rep packets from its neighbours over a certain time interval. All nodes that receive the Check packet from n_2 send a Rep packet to n_2. Now, node n_2 will compare all the received version numbers obtained from the Rep packets.

If all the received version numbers are equal and match its own version number, then n_2 sends a Hello packet to n_1, stating that the correct version number is that held by n_2 and that it has the updated code too. If the received version numbers from the Rep packets are the same but differ from that of n_2, n_2 downloads the available code fragment by downloading from one of the Rep senders. After downloading the code, n_2 will eventually send Hello messages. Further, if there are only two different version numbers received, then n_2 chooses the higher one (as there are only two versions in the network during the dissemination process).

On the other hand, if n_2 obtains more than 2 version numbers, then this indicate an error in the network. For any node n_3 that sent Rep packets to n_2 with version numbers that violate the consistency predicate, n_2 send Prob packets to n_3. These nodes, in turn, send check packets to their neighbours and the process is repeated.

Once the recovery process reaches a region outside of the fault-affected area, nodes on the border of the fault-affected area will receive Rep packets with at most two different version numbers (this is the case when only part of a neighbourhood has been updated). Once a

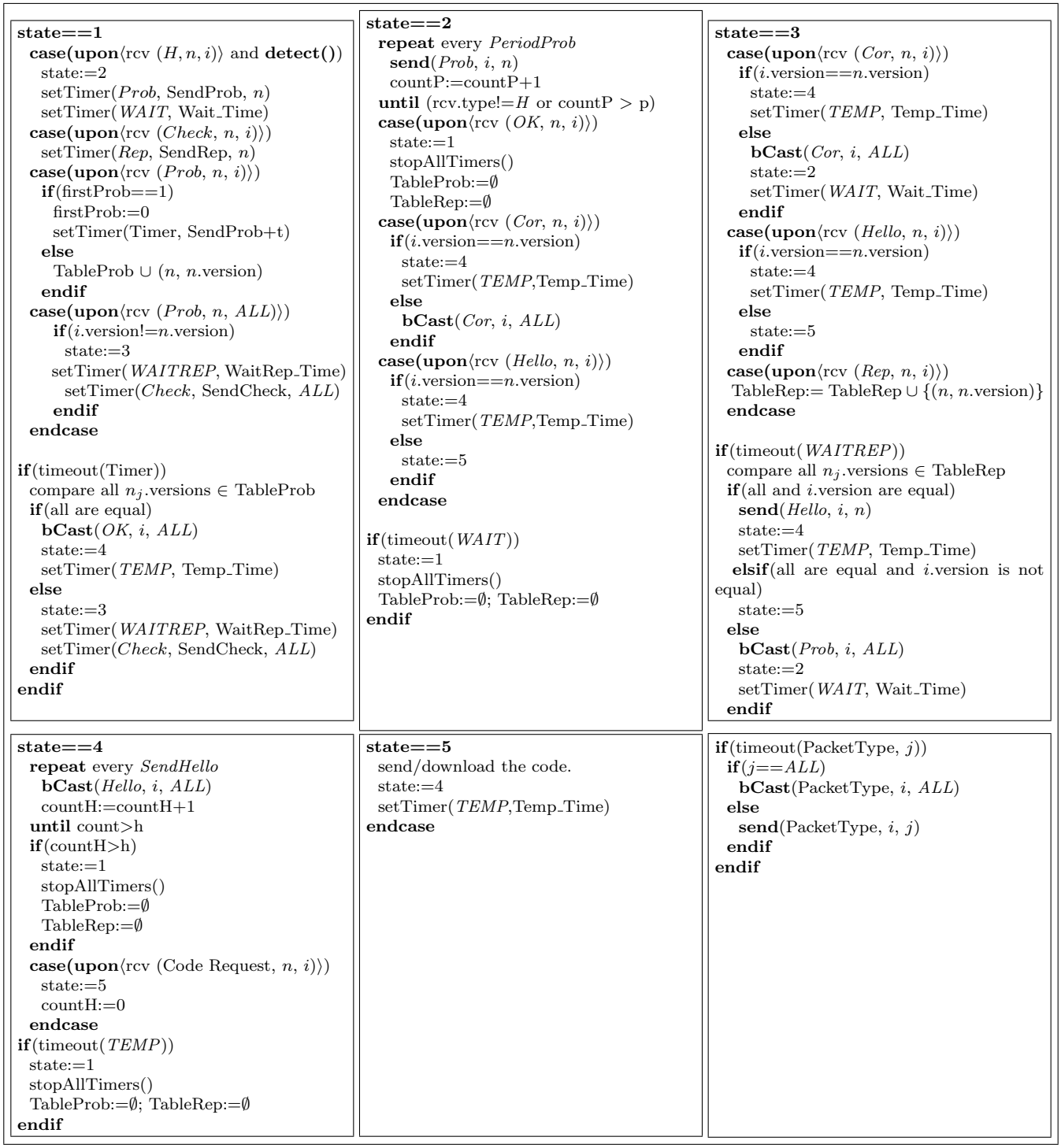


FIGURE 3. BestEffort-Repair Algorithm.

node receiving these Rep packets decides on the correct version number, it broadcasts a Hello message (after possibly downloading the associated code, if it does not already have it) a few times to ensure that its neighborhood learns about the correct version.

Because of reasons such as transient link failures, a Prob packet sent by n_1 may not be received by n_2. To overcome this issue, n_1 periodically sends a Prob packet to n_2 for some predefined times until n_1 receives an implicit acknowledgement packet, such as Hello or Check.

The Consistent-Repair protocol, when added to a fault-intolerant code dissemination protocols, transforms the protocol into a non-masking fault-tolerant one that satisfies the Consistent CD specification. It leverages the fact that, at any time during the new code dissemination, there will be at most two codes in the network: (i) the old one and (ii) the new one. This means that any node only need to know about two different version numbers. Once a node knows about these, it

Variables of process i:
firstHello:{(id,version):id∈ ℕ,version∈ ℕ}

state==2

```
case(upon⟨rcv (Hello, n, i)⟩)
  if(firstHello.version==∅)
    firstHello.version:=n.version
    firstHello.source:=n
  elseif(firstHello.version > n.version)
    if(i.version==firstHello.version)
      state:=4
      setTimer(TEMP,Temp_Time)
    else
      state:=5
    endif
  elseif(firstHello.version < n.version)
    if(i.version==n.version)
      state:=4
      setTimer(TEMP,Temp_Time)
    else
      state:=5
    endif
  endif
endcase

if(timeout(WAIT))
  if(firstHello.version==∅)
    state:=1
    stopAllTimers()
    TableProb:=∅; TableRep:=∅
  else
    if(i.version > firstHello.version)
      state:=4
      setTimer(TEMP, Temp_Time)
    else
      state:=5
    endif
  endif
endif
```

state==3

```
case(upon⟨rcv (Rep, n, i)⟩)
 TableRep:= TableRep ∪ {(n, n.version)}
endcase

if(timeout(WAITREP))
  if(detect())
    bCast(Prob, i, ALL)
    state:=2
    setTimer(WAIT, Wait_Time)
  elseif(i.version < Table.version))
    state:=5
  else
    state:=4
    setTimer(TEMP, Temp_Time)
  endif
endif
```

FIGURE 4. Consistent-Repair Algorithm.

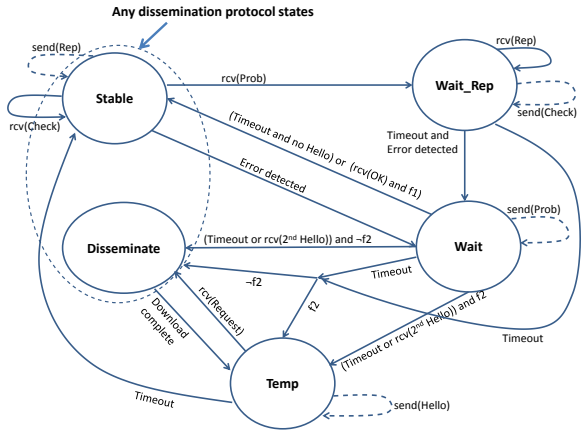


FIGURE 5. The state machine for Consistent-Repair. The two states in dashed circle are the states of any code dissemination protocol. f1=TRUE if the sender of an *OK* packet is the node which sent an *H* packet, f2=TRUE if a node is updated, FALSE otherwise.

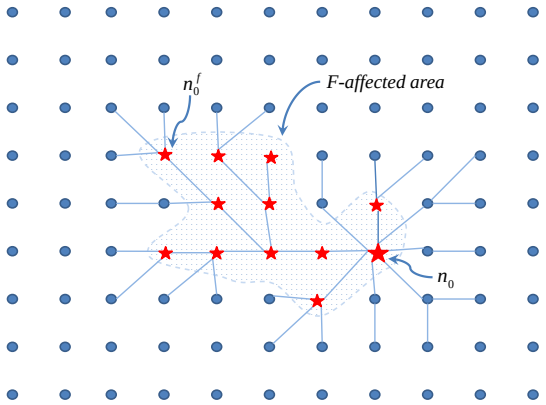


FIGURE 6. An example of a fault-affected area.

can choose the higher one, which is associated with the updated code.

The Consistent-Repair protocol is shown in Figure 4, which shows the code for when the process is in state 2 (Wait) and state 3 (Wait-Rep) is shown (the code for when the process is in state 1, 4 and 5 is the same as for BestEffort-Repair (Figure 3)). On the other hand, Figure 5 illustrates the state machine of the Consistent-Repair protocol.

An example of a fault-affected area is depicted in Figure 6, and the corresponding correction tree for the fault-affected area, as constructed by Consistent-Repair, is depicted in Figure 7.

We prove an important property of Consistent-Repair, in that Consistent-Repair generates a correction tree of depth at most $f+2$, where f is the diameter of the F-affected area.

Lemma 6.2 (Correction Tree). *Given a network $G=(V,A)$, a detectable fault model F, and an F-affected area G', with the diameter of the area being f. Then, Consistent-Repair constructs a tree of depth at most $f+2$ rooted at the node that first detects an error.*

Proof:

Assumptions: We denote a node that first detects an error by n_0, and we denote a node a distance d from n_0 by n_0^d. We assume that node n_0 detects an error after receiving a packet from some node n_0^1, and that n_0 is on the boundary of the F-affected area (some of its neighbours are F-affected, some are not), and $f>1$.

According to Consistent-Repair, n_0 will send a Prob packet to n_0^1 and then goes to the Wait state. This starts

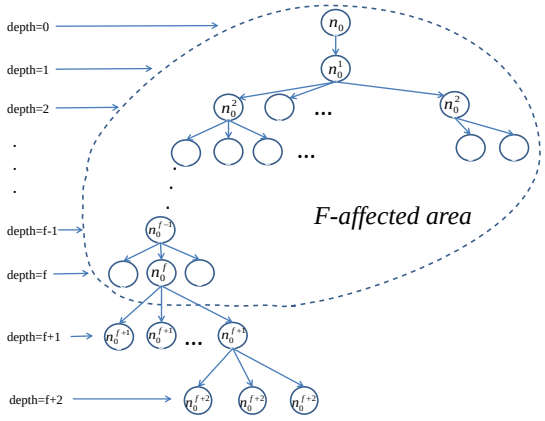


FIGURE 7. Correction Tree Constructed by Consistent-Repair

a graph with n_0 as the root at depth=0 (see Figure 7). Node n_0^1, the child of n_0, has depth = 1. Node n_0^1, in turn, broadcasts Check packets to its neighbours. Node n_0^1 then goes to the WaitRep state to wait for Rep packets from the informed neighbours, which are at depth=2.

Now, for node n_0^1, since the diameter of the F-affected area is f, this means that n_0^1 will eventually send Prob packets to all senders of faulty Rep packets. We focus on one such node, which we denote by n_0^2. This process spawns a (sub)tree, with n_0^1 as the root of the subtree, and node n_0^2 will send Check packets to its neighbours (see Figure 7). When the node n_0^f (at the other end of the F-affected area is reached), in the worst case, it will detect faulty Rep packet from at least one node, which we denote by n_0^{f+1} (at a distance of $f+1$ from n_0). It will then send a Prob packet to n_0^{f+1}, which, in turn, sends Check packets to its neighbours at a distance of $f+2$. Since the F-affected area is of diameter f, all of the Rep packets to node n_0^{f+1} will hold at most two version numbers (and the tree does not grow anymore). At this point, node n_0^{f+1} can decide on an appropriate version number. Hence, the tree is of a depth of at most $f+2$. □

In effect, when a node sends a Prob or Check packet, new subtrees are created, and the depth of the tree increases by 1. When a node receives identical information from its children, it sends a Hello packet to its parent, indicating that it has the associated code. At this point, the dependency of its children ends, reducing the depth of the tree by 1. It should be further noted that a tree is constructed for every node that detects an error. So, at any point in time, there may be several correction trees in the network.

We now prove the correctness of Consistent-Repair.

THEOREM 6.2 (f-local correction). *Given a network $G = (V, A)$, a detectable fault model F, and an F-affected area G' of diameter f. Then, Consistent-Repair guarantees that, eventually, all nodes in G' will have a state consistent with their neighbourhood.*

Proof.

We will prove by induction on the correction tree (see Lemma 6.2) that node n_0, and all nodes in G', will eventually download the correct code.

Assumptions: (i) We assume a node n_0 has downloaded the stale code, (ii) node n_0 has detected an error (state inconsistency) (i.e., node n_0 is the root of the correction tree). We will denote a node at a distance d from another node n_0 by n_0^d.

Base case:

We prove for the case of a node, which we denote by n_0^{f+1} at depth $= f+1$ (i.e., the last rooted subtree). Node n_0^{f+1} will eventually receive a set of Rep packets with identical version numbers. Node n_0^{f+1} will then eventually download the correct code from one of the Rep packets senders.

Inductive hypothesis:

Assume that a node n_0^i, where $0 < i \leq f$, eventually receives a *Hello* packet from a node n_0^{i+1} and then updates its code.

Inductive Step:

We need to prove that a node n_0^{i-1}, a neighbour node of n_0^i, eventually receives a *Hello* packet and updates its code.

In Consistent-Repair, node n_0^i will broadcast *Hello* packets periodically up to h times after receiving a *Hello* packet or having updated its code. If node n_0^{i-1} receives a *Hello* packet from n_0^i, n_0^{i-1} will update its code from one of the n_0^i nodes, which proves the inductive step. Else, if due to message losses, n_0^{i-1} does not receive a *Hello* packet from n_0^i, then, node n_0^{i-1} waits *Wait_Time* and goes to PS state and operates normally. Eventually, node n_0^{i-1} or a neighbour node of n_0^{i-1} will detect the error, and executes Consistent-Repair again. Assuming that the number of message losses is finite, eventually, node n_0^{i-1} will eventually get a *Hello* packet, when n_0^{i-1} can download the code from the node it receives the Hello packet from. □

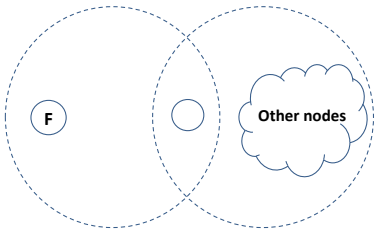


FIGURE 8. Second scenario: Topology where a faulty node has only one neighbour.

7. EXPERIMENTAL SETUP AND RESULTS

In this section, we present the deployment and simulation setup used to evaluate the working and

performance of both BestEffort-Repair and Consistent-Repair. We subsequently present the results of each experiment for both protocols. The setup was identical for both protocols.

7.1. Deployment and Simulation Setup

Deployment To assess the proper working of both protocols, we conducted both a deployment and simulation experiments. For the deployment experiments, we implemented both protocols on TinyOS-2.1 [47] and performed indoor experiments using CM5000 sensor motes, which are based on the TelosB platform design. We used 10 motes in our deployment experiments, setting the transmission power of each node to a very low level 2.

Simulation To evaluate the overhead of both protocols in large scale networks, we conducted simulation experiments using TOSSIM [19] as simulator. The network topology used is a 20*20 grid, with the distance between two nodes set at around 10 ft, with nodes having a communication radius of 30 ft. The network topology with asymmetric links is constructed by a tool given on tinyos.net. Each node is given a noise model from the heavy-meyer noise trace file located in Tossim/noise folder.

The parameter values for the various timers of both protocols used in our deployment and simulation experiments 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 a *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 the *Temp* state from the *Wait* state where it waits for a shorter time. The only case when a node waits for *Wait_Time* is when there is a packet loss. *Temp_Time* time is independent of other parameters. The value of t should be small because a node waits a maximum of *SendProb* time units to receive all possible *Prob* packets. The values of h and p can be set to any value.

In our experiments, each node periodically broadcasts an application packet (or any other traffic that drives the dissemination) H, with the period randomly selected between $[0, U]$ at the start.

Wait_Time	50 (30) sec	*SendRep*	2 sec
SendCheck	2 sec	*WaitRep_Time*	4 sec
SendHello	1.5 sec	*Temp_Time*	30 (20) sec
SendProb	1 sec	t	0.2 sec
PeriodProb	7 sec	p	5
U	60 (1) sec	h	2

TABLE 1. Parameter values used in simulation and deployment experiments (deployment values are within brackets).

7.2. Scenarios: Simulation and Experiments

7.2.1. Simulation Scenarios

In our simulations, we simulated two scenarios: (i) **Scenario 1**: we varied the number of corrupted nodes per circular area, which has diameter of 60 feet (varying the fault density), and (ii) **Scenario 2**: 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 nodes to be corrupted were selected randomly in the given area. We then counted (i) the number of Repair packets (BestEffort-Repair or Consistent-Repair) sent, (ii) the number of involved nodes, i.e., nodes that sent at least one 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 the simulations 5 times and computed the min, average and max values over the 5 runs.

7.2.2. Experimental Scenarios

Our implementation of BestEffort-Repair takes 222 bytes of memory, which includes two tables (TableProb and TableRep - see Figure 2), each of size 50 entries of 2 bytes each, and other algorithm related variables. Depending on the size of the network, the size of the tables can be varied. On the other hand, our implementation of Consistent-Repair takes 144 bytes, which includes 3 arrays of size 3*2, with each entry of size 2 bytes.

Our claim is that both BestEffort-Repair and Consistent-Repair can help any code dissemination protocol that has enough state to enable the detection of an erroneous state to eventually guarantee that every node has the updated code. As a result, we tested both BestEffort-Repair and Consistent-Repair by adding them to Varuna [4], one of the latest code dissemination protocols, on three different network topologies under four different scenarios. In the first scenario, the network was complete where all nodes could communicate with each other. In the second scenario, the network topology was formed by placing a faulty node at one end of the network in such a way that it has only one neighbour, with the network remaining connected and multi-hop (See Figure 8). In the third scenario, the topology was formed by randomly deploying the nodes such that the network is connected and multi-hop. In these three scenarios, only one node is used as a faulty node with a corrupted version number or corrupted neighbourhood table, since Varuna's state consists of a neighbourhood table and a variable holding the version number. Finally, in the fourth scenario, the network topology that is used is the same as that used in scenario three, but with two

faulty nodes: one with the version number corrupted and the other with the neighbourhood table corrupted.

Faults were artificially injected in Varuna by changing the version number and/or neighbourhood table entries of the faulty nodes. Recall that both BestEffort-Repair and Consistent-Repair is executed only when an error (i.e., erroneous state) is detected, so the faults injected were such that variables were modified in such a way to trigger an error that will be detected, leading to the execution of *Repair*. In all tests, the faulty node(s) were booted after all correct nodes were successfully booted so as to assess the impact of faulty nodes on a dissemination process that is already in progress.

7.3. BestEffort-Repair

7.3.1. Simulation Results

Number of nodes: From Figure 9(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 in the stabilisation process is proportional to the size of the corrupted area. Further, in Figure 9(b), we observe that, as the size of the area is increased (i.e., fault density decreases), the number of nodes executing the protocol becomes almost constant, on the average. This is because, with decreasing fault density, most faults tend to appear as a single independent fault, with each of them involving a similar number of nodes, and may only involve at most their 2-hop neighbourhood. This implies that, in general, BestEffort-Repair tends to access only a bounded neighbourhood (similar to Consistent-Repair).

Number of packets: We notice a similar trend in Figure 10 that supports the observation that, in general, BestEffort-Repair tends to access a bounded neighbourhood. In Figure 10(a), we observe that the number of *Repair* packets sent varies linearly with the number of corrupted nodes. Since the number of *Repair* messages sent is much less than the size of the network, it implies that only part of the network was involved in the stabilisation process.

The discrepancy between the maximum number and minimum number of nodes or *Repair* packets is often due to the link quality, making retransmissions necessary.

7.3.2. Experimental Results

In our experiments, we measured (i) the number of transmitted *Repair* packets and (ii) the latency required to correct the error. For each scenario, we ran *Repair* 20 times and computed the average of the *Repair* packets and latency.

As expected, in all of our experiments, adding BestEffort-Repair to Varuna ultimately corrected the errors. The results obtained for scenarios $1 \ldots 4$ are shown in Tables 2, 3, 4, and 5 respectively. In all cases, the average number of *Repair* packets and the latency are reasonably low. For example, focusing on Table 2, the minimum number of *Repair* packets is proportional to the size of the neighbourhood of the faulty node. This is as expected since most nodes are expected to send *Rep* packets due to the network being complete. The difference between minimum and maximum values is due to the loss characteristics of the wireless medium. For example, in the case (see Table 5) where the highest latency among all experiments was 218, 218 milliseconds and the largest number of packet transmissions was 82 can show this property. In that particular case, Prob packets sent by a node that detected the error was not received by a receiver. Therefore, the node had to retransmit Prob packets and the number of Prob packet retransmissions was 29 and the time taken for that was about 155, 000 milliseconds.

7.4. Consistent-Repair

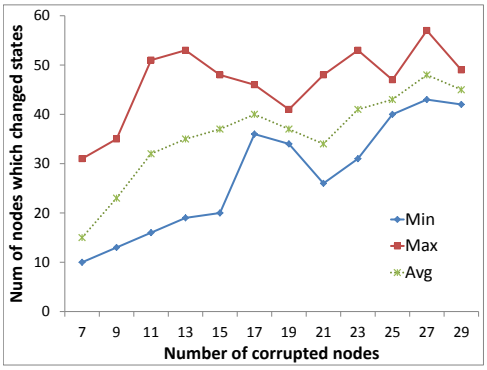
7.4.1. Simulation Results

We now present the result of simulation experiments of Consistent-Repair.

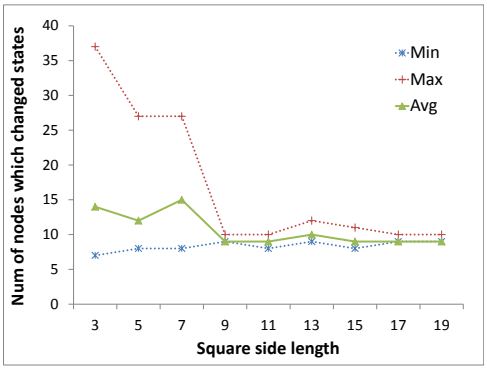
Number of nodes: From Figure 11(a), we observe that, on average, the number of nodes executing the Consistent-Repair varies linearly with the number of corrupted nodes. Given that the number of nodes involved is much less than the size of the network, it implies that the number of nodes involved in the stabilisation process is proportional to the size of the corrupted area. We also observe, in Figure 11(b), that, as the size of the area within which faults occur is increased (i.e., fault density decreases), the number of nodes executing the protocol becomes almost constant, on the average. This is because, with the decreasing fault density, most faults tend to appear as single independent faults, i.e., the fault-affected area is of size 1. Each corrupted node may only involve at most their 2-hop neighbourhood during recovery. These two observations support the fact that Consistent-Repair in f-local, with f being the diameter of the fault-affected region.

Number of packets: We observe a similar trend in Figure 12 that supports the f-locality property of Consistent-Repair. In Figure 12(a), we observe that the number of *Repair* packets sent varies linearly with the number of corrupted nodes. Since the number of *Repair* messages sent is much less than the size of the network, it implies that only part of the network was involved in the stabilisation process.

As in the case with BestEffort-Repair, the discrepancy between the maximum number and minimum

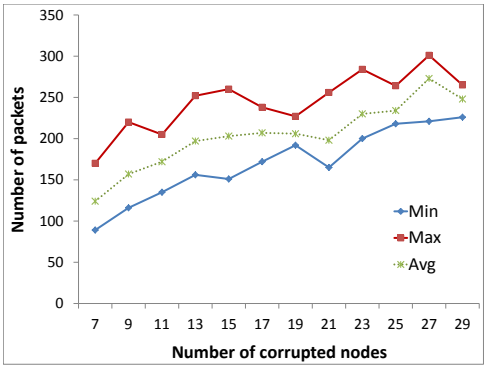


(a) Scenario 1: Num. of nodes executing BestEffort-Repair vs Num. of Corrupted Nodes, Area diameter = 60ft

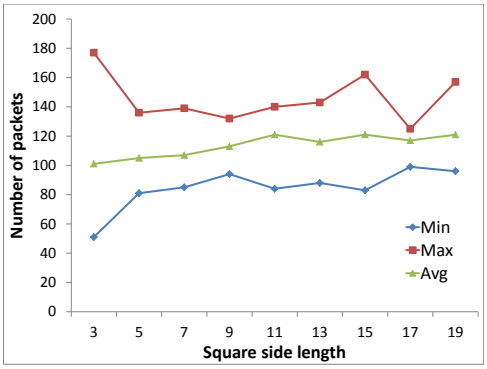


(b) Scenario 2: Num. of nodes executing BestEffort-Repair for 5 corrupted nodes per sq. area.

FIGURE 9. Maximum, minimum and average number of nodes executing BestEffort-Repair



(a) Scenario 1: Num. of packets vs Num. of corrupted nodes, Area diameter = 60ft



(b) Scenario 2: Num. of packets for 5 corrupted nodes per sq. area.

FIGURE 10. Maximum, minimum and average number of transmitted BestEffort-Repair packets

number of nodes for Consistent-Repair packets is often due to the link quality, making retransmissions necessary.

7.4.2. Experimental Results

In our experiments, we measured (i) the number of transmitted *Repair* packets and (ii) the latency required to correct the error. For each scenario, we ran *Repair* 20 times and computed the average of the *Repair* packets and latency.

As expected, in all of our experiments, adding Consistent-Repair to Varuna ultimately corrected the errors. The results obtained for scenarios 1...4 are shown in Tables 6, 7, 8, and 9 respectively. In all cases, the average number of *Repair* packets and the latency are reasonably low. For example, focusing on Table 6, the minimum number of *Repair* packets is proportional to the size of the neighbourhood of the faulty node (as in the case for BestEffort-Repair). This is as expected since most nodes are expected to send *Rep* packets due to the network being complete. The difference between minimum and maximum values is due to the loss characteristics of the wireless.

7.5. Differences Between BestEffort-Repair and Consistent-Repair

From Figures 9 to 10 (for BestEffort-Repair) and Figures 11 to 12 (for Consistent-Repair), it can be observed that, in general, Consistent-Repair involves more messages and nodes. This is due to the fact that, given that Consistent-Repair makes more informed decisions to prevent any erroneous downloads, more nodes are involved and, thus, they send more messages. On the other hand, given that BestEffort-Repair is biased towards fast recovery, it attempts to make the network state consistent again, even if erroneous downloads are involved.

It can also be noted from Tables 2 to 5 (for BestEffort-Repair) and Tables 6 to 9 (for Consistent-Repair) that (i) the best case for BestEffort-Repair (i.e., minimum

Scenario 1	Version corrupted			Table corrupted		
	Min	Max	Avg	Min	Max	Avg
Number of packets	8	23	12.6	7	14	12.4
Time(millisec)	6962	17544	8070	6011	14449	8791

TABLE 2. Scenario 1, BestEffort-Repair: Complete network

Scenario 2	Version corrupted			Table corrupted		
	Min	Max	Avg	Min	Max	Avg
Number of packets	10	39	19	9	39	18
Time(millisec)	5497	111142	31518	4719	83760	15013

TABLE 3. Scenario 2, BestEffort-Repair: Faulty node at one end of the network

values) is, in general, better than that of Consistent-Repair and (ii) the worst case for BestEffort-Repair (i.e., maximum values) is, in general, worse than that of Consistent-Repair. As mentioned earlier, BestEffort-Repair can, in the worst case, involve the whole network during recovery, as opposed to Consistent-Repair which will only involve its $f + 2$ hop neighbourhood. In the best case, BestEffort-Repair may receive the proper Hello message first and helps the affected area to receiver quickly, whereas Consistent-Repair will wait for several messages to arrive before reaching the decision.

8. CASE STUDY: ADDING BESTEFFORT-REPAIR AND CONSISTENT-REPAIR TO VARUNA

In this section, we discuss the addition of BestEffort-Repair and Consistent-Repair to Varuna [4]. The reason for choosing Varuna is that it is one of the latest code dissemination protocols that have been proposed. The code that was used in the deployment was reused for the simulation experiments described in this section.

As mentioned before, both protocols are triggered by the detection of an error in the state of the code dissemination protocol, in this case Varuna. In Varuna, such a detection is enabled by one of the following conditions: (i) two nodes' version numbers are corrupted in such a way that the difference in versions is strictly 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 of the message exists in its neighbourhood table. Also, we disallow faults, under the detectable fault model, that cause old code to appear as new and new as old (i.e., all updated nodes have old version numbers and non-updated nodes have the new version number). Also, this disallows nodes to be corrupted in identical ways.

We simulated the composite protocol of Varuna and BestEffort-Repair and Consistent-Repair in TOSSIM. All nodes, except faulty nodes, are booted in the first minute. Faulty nodes are located at the center of a grid network of 20*20. A packet with new version number is injected after 2 minutes. We simulated three faulty scenarios: (i) with 1 fault, (ii) with 4 faults and (iii) with 7 faults. For each faulty scenario, we booted the faulty nodes (i) 30 seconds, (ii) 45 seconds, and (iii) 60 seconds. This is so that only a proportion of nodes has the updated code version. The reason for booting faulty nodes some time after the updated code is injected is to ensure that nodes that have the stale code are chosen to have faults injected into them. 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 7. Further, the values for Varuna-specific parameters are: DISS-RAND=2 sec, ADV-RAND=2 sec, τ=8 sec, T_{MOODY}=1 min.

8.1. Performance of Best-Effort Repair

Special Case

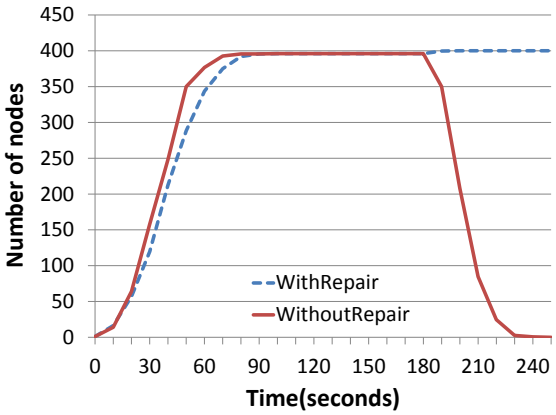


FIGURE 13. 4 faulty nodes booted 180 seconds after updated code injection.

We first start with a special case where simulated the situation where, due to situations such as duty cycling, some nodes may have been sleeping, missing the code update. In Figure 13, 4 such nodes are booted 180 seconds after the code update has been injected into the network. Further, these 4 nodes are faulty as well. We observe that, in Varuna, all nodes in the network eventually end up downloading the stale code, while the composite protocol of Varuna and either Repair protocol ensures that the whole network has the updated code.

From Figures 14, 15 and 16, we make two important

Scenario 3	Version corrupted			Table corrupted		
	Min	Max	Avg	Min	Max	Avg
Number of packets	16	45	24	10	26	24
Time(millisec)	15745	124786	50472	10705	144214	37097

TABLE 4. Scenario 3, BestEffort-Repair: Connected Random graph, 1 fault

Scenario 4. Version and table corrupted			
	Min	Max	Avg
Number of packets	15	82	27
Time(millisec)	14134	218218	59231

TABLE 5. Scenario 4, BestEffort-Repair: Connected Random graph, two faults

observations: (i) In all cases, injecting transient faults in the network during Varuna execution causes the whole network to disseminate stale code. This shows that Varuna cannot handle transient faults. On the other hand, when BestEffort-Repair is added to Varuna, every node eventually downloads the correct code.

8.1.1. Packet Overhead

In Figure 17(a), it can be seen that the packets overhead induced by Repair on Varuna is low. Specifically, with 4 faulty nodes, the packet overhead is 0.4% while, with 7 faulty nodes, the packet overhead is less than 3%. From Figure 10, it can be observed that the number of Repair 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).

8.1.2. Temporal Overhead

In Figure 17(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 BestEffort-Repair on Varuna*, highlighting the fact that BestEffort-Repair is biased towards fast recovery.

8.2. Performance of Consistent-Repair

From Figures 18, 19 and 20, we make one important observation: When Consistent-Repair is added to Varuna, every node eventually downloads the correct code.

8.2.1. Packet Overhead

In Figure 21, it can be seen that the packets overhead induced by Consistent-Repair on Varuna is very low. Specifically, with 4 faulty nodes, the packet overhead is less than 0.8% (see Figure 21). From Figure 12, it can be observed that the number of Repair 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).

8.2.2. Temporal Overhead

In Figure 17(b), 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 90-100 seconds. Thus, *there temporal overhead induced by Consistent-Repair on Varuna is approximately 10%–20%.* This supports the fact that Consistent-Repair needs more time for informed decisions (as opposed to BestEffort-Repair).

8.3. Difference Between BestEffort-Repair and Consistent-Repair

Up to now, we have observed that when adding BestEffort-Repair and Consistent-Repair to Varuna, all nodes eventually download the updated code, meaning that both of them are correctors for strong CD. It has been shown that the temporal overhead induced by BestEffort-Repair on Varuna is lower than that of Consistent-Repair as well as the packet overhead of BestEffort-Repair (0.4%) on Varuna is roughly 2 times as low as Consistent-Repair (0.75%). This is due to the fact that BestEffort-Repair is biased towards fast recovery, requiring less packets. On the the other

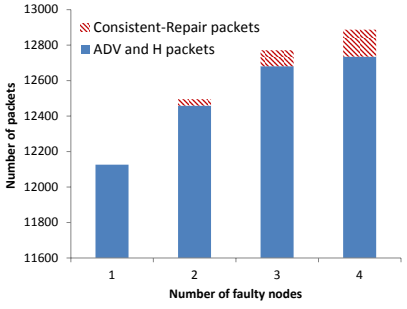
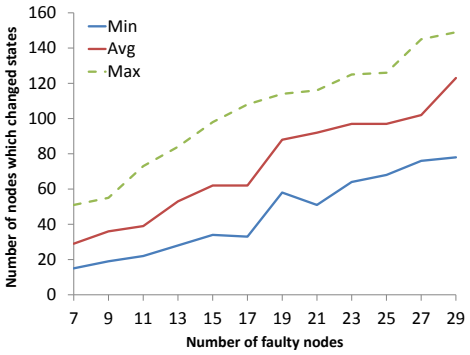
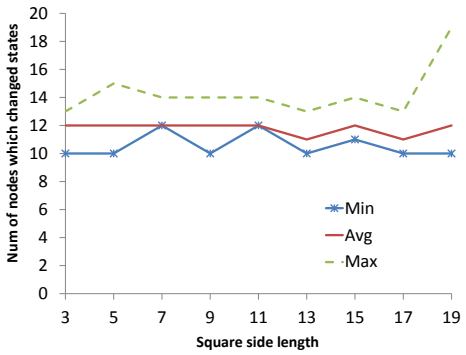


FIGURE 21. Number of (ConsistentRepair + ADV) packets sent vs number of corrupted nodes

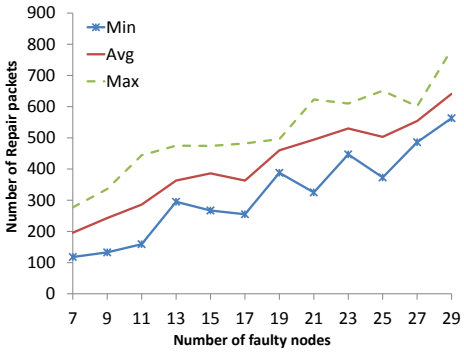


(a) Scenario 1: Num. of nodes executing Consistent-Repair vs Num. of Corrupted Nodes, Area diameter = 60ft

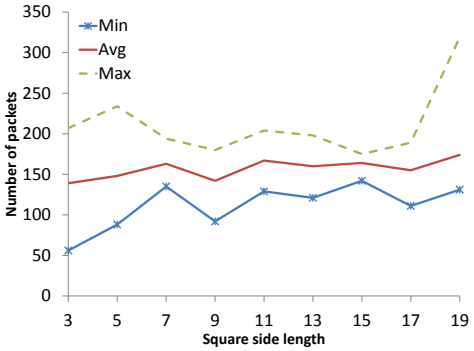


(b) Scenario 2: Num. of nodes executing Consistent-Repair for 5 corrupted nodes per sq. area.

FIGURE 11. Maximum, minimum and average number of nodes executing Consistent-Repair



(a) Scenario 1: Num. of packets vs Num. of corrupted nodes, Area diameter = 60ft



(b) Scenario 2: Num. of packets for 5 corrupted nodes per sq. area.

FIGURE 12. Maximum, minimum and average number of transmitted Consistent-Repair packets

hand, we motivated Consistent-Repair to allow for more informed recovery in that it reduces the number of erroneous downloads (where an updated node ends up downloading the old code to eventually update again). In this respect, in our experiments, we observed that, on average, BestEffort-Repair causes 5 erroneous downloads - which is allowed under the BestEffort CD specification (to eventually download the correct code), whereas, with Consistent-Repair, there were no erroneous downloads.

9. DISCUSSION

In this section, we discuss some issues arising from our approach in a Q&A style.

- Was it not possible to design generic detectors, similar to the design of generic correctors proposed in the paper?
  We have addressed the problems caused by corrupted code dissemination program state induced by transient faults. We have proposed two generic corrector protocols that correct the inconsistency in code dissemination protocols, and are triggered by protocol-specific detectors. This means that, for such detectors, an in-depth understanding of the protocol involved is needed. It may be argued that a generic detector can be used. For example, it may be suggested that a hash of the code and version number is piggybacked onto messages. A receiver node can then compute its own hash and version number and then compare to detect errors. However, the above scheme will not work for a number of reasons:

  1. Our fault model assumes that transient faults will corrupt the state of the code dissemination protocol, as opposed to corrupting the code itself.
  2. As the code dissemination protocol executes, different nodes will have different version numbers (old or new). Then, using the hash of the code and version number will not work.
  3. Corruption of variables other than the version numbers can still lead to the code dissemination protocols not working properly (e.g., in Varuna, corrupting the neighbourhood table caused problems), and is not captured by the above scheme.

Scenario 1	Version corrupted			Table corrupted		
	Min	Max	Avg	Min	Max	Avg
Number of packets	12	29	16.2	14	30	16.3
Time(millisec)	30137	30385	30351	30372	30383	30375

TABLE 6. Scenario 1, Consistent-Repair: Complete network

Scenario 2	Version corrupted			Table corrupted		
	Min	Max	Avg	Min	Max	Avg
Number of packets	13	24	15.6	11	36	16.5
Time(millisec)	30179	60937	31467	30183	112728	37304

TABLE 7. Scenario 2, Consistent-Repair: Faulty node at one end of the network

4. Further, we also believe such an approach to be unsuitable for WSNs, given that specific hardware is needed for encoding and decoding the checksums, though such hardware is cheap.

What this means is that, in general, a generic detector would miss some errors, leading to erroneous downloads of code or, in the worst case, the whole network having the old code. In fact, it was suggested in [48] that the efficiency of generic detectors tend to be not very high [48], meaning that these detectors can have a high false positive, thereby triggering the corrector protocols unnecessarily, or false negative rate, whereby codes may download the wrong code.

- The two protocols are quite similar. Why do we need both of them?
  It can be argued that the protocols are quite similar to each other. Though the differences are not major, they are geared for specific situations. Specifically, BestEffort-Repair is best used when a corrupted node is surrounded by neighbours with the same version number (old or new), while Consistent-Repair is better suited when a corrupted node has at least one updated node nearby. As has been shown, Best-Effort Repair completes faster than Consistent-Repair at the expense of more redundant downloads. For example, if a bug has been detected and the program needs to be updated very quickly, BestEffort-Repair is more suitable. On the other hand, if the program needs updated (due to, say, a change in requirements), then Consistent-Repair will be more appropriate. Nevertheless, as future work, we plan on investigating the design of an adaptive code dissemination protocol based on the position of a corrupted node in the network. In such a case, in a fault-affected area, some nodes will execute BestEffort-Repair and others Consistent-Repair. Thus, since BestEffort-Repair is geared towards fast recovery, the fact that it is surrounded by nodes with the same version number makes it state consistent during recovery. Further, Consistent-Repair guarantees a f-local correction (f being the diameter of the affected area), while BestEffort-Repair is a global algorithm, involving, in the worst case, the whole network. However, in the general case, BestEffort-Repair rarely needs the whole network for correction, requiring the $O(f)$-hop neighbourhood.
- How realistic is the detectable fault model?
  The detectable fault model disallows two types of fault actions: (i) nodes in a neighbourhood to be identically corrupted and (ii) for nodes with the old (resp. new) code to appear as new (resp. old). Though these faults can actually in practice, the probability of such faults to occur is very low. If these faults do occur, then it is impossible to guarantee that dissemination of the new code will terminate properly. One way to circumvent this problem is to require the sink to query the network after some time to determine if nodes have the proper code, using possibly a hash of the code and the version number.
- Are the correctors themselves resilient to transient faults?
  The theory of detectors and correctors [17] points out that correctors are themselves non-masking fault-tolerant to transient faults, i.e,, they can temporarily violate their safety specification but will satisfy their liveness. So, the two correctors we proposed are non-masking fault-tolerant. This means that it may cause nodes to download erroneous code or have the wrong version numbers but eventually, when faults stop, all the nodes will have the correct versions.
- Will the corrector protocols developed in this paper work for selective reprogramming dissemination protocols?
  We believe the answer to be positive. However, there are some preprocessing to be done before they can be applied. Specifically, in WSNs, the network is connected but, for selective reprogramming, the selected nodes may not form a connected (sub)network. Thus, a logical network needs to be built between these nodes, as a kind of network overlay, using techniques such as [49]. Then, the algorithms can be applied on top of the overlay.

10. CONCLUSION

In this paper, we have addressed the problem of code dissemination in the presence of transient faults that

Scenario 3	Version corrupted			Table corrupted		
	Min	Max	Avg	Min	Max	Avg
Number of packets	9	32	16.2	9	28	15.5
Time(millisec)	30336	30429	30374	30100	30391	30288

TABLE 8. Scenario 3, Consistent-Repair: Connected Random graph, 1 fault

Scenario 4. Version and table corrupted			
	Min	Max	Avg
Number of packets	12	42	28
Time(millisec)	30545	61003	43498

TABLE 9. Scenario 4, Consistent-Repair: Connected Random graph, two faults

corrupt the state of the code dissemination program. We have provided three local specifications, namely (i) strong CD (ii) BestEffort CD and (iii) Consistent CD. We have proved some impossibility results and have provided two generic corrector protocols, viz. BestEffort-Repair and Consistent-repair, that can be added to any fault-intolerant code dissemination protocol to transform them into non-masking fault-tolerant code dissemination programs. We have conducted in-depth experiments with both corrector protocols: We have shown that BestEffort-Repair allows for fast recovery but can induce erroneous downloads (i.e., an updated node may download an old code). On the other hand, Consistent-Repair only enables consistent downloads. The packet overhead of each protocol has also been shown to be very low. The main contribution of our paper is the automated transformation of a whole class of fault-intolerant code dissemination protocols (such as Varuna, Trickle etc) into non-masking fault-tolerant protocols.

REFERENCES

[1] Lanigan, P. E., Gandhi, R., and Narasimhan, P. (2005) Disseminating code updates in sensor networks: Survey of protocols and security issues. Technical Report CMU-ISRI-05-122. Institute for Software Research, Carnegie Mellon University.

[2] Kulkarni, S. and Wang, L. (2009) Energy-efficient multihop reprogramming for sensor networks. *ACM Transactions on Sensor Networks*, **5**, 16:1–16:40.

[3] Levis, P., Patel, N., Culler, D., and Shenker, S. (2004) Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. *In Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI*, pp. 15–28.

[4] Panta, R. K., Vintila, M., and Bagchi, S. (2010) Fixed cost maintenance for information dissemination in wireless sensor networks. *Proc. SRDS*, pp. 54–63.

[5] Mottola, L. and G.P. Picco, A. A. S. (2008) Figaro: Fine-grained software reconfiguration for wireless sensor networks. *Proceedings of EWSN* Lecture Notes in Computer Science, pp. 286–304. Springer-Verlag.

[6] Ni, K. and et al. (2009) Sensor networks data fault types. *Transactions on Sensor Networks*, **5**.

[7] Finne, N., Eriksson, J., Dunkels, A., and Voigt, T. (2008) Experiences from two sensor network deployments self-monitoring and self-configuration keys to success. *Proc. of Int. Conf. on Wired/Wireless Internet Communications (WWIC).*

[8] Werner-Allen, G., Lorincz, K., Johnson, J., Lees, J., and Welsh, M. (2006) Fidelity and yield in a volcano monitoring sensor network. *Proc. of 7^{th} Symp. on Operating Systems Design and Implementation (OSDI).*

[9] Jhumka, A. and Mottola, L. (2013) Neighborhood monitoring and view consistency enforcement in wireless sensor networks. Technical Report TR2013.1321. Politecnico di Milano.

[10] Hui, J. W. and Culler, D. (2004) The dynamic behavior of a data dissemination protocol for network programming at scale. *Proceedings of the 2nd international conference on Embedded networked sensor systems*, New York, NY, USA SenSys '04, pp. 81–94. ACM.

[11] Bapat, S. and Arora, A. (2006) Stabilizing reconfiguration in wireless sensor networks. *Proc. of IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC) 2006)*, pp. 52–59.

[12] Reason, J. M. and Rabaey, J. M. (2004) A study of energy consumption and reliability in a multihop sensor network. *ACM Mobile Computing and Communications Review*, **8**, 84–97.

[13] Crossbow Technology, I. (2003) Mote in-network programming user reference, www.tinyos.net/tinyos-1.x/doc/xnp.pdf.

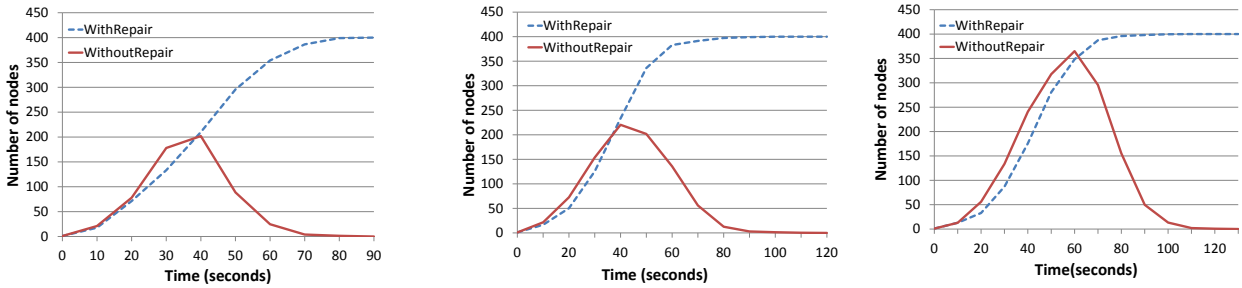
[14] Levis, P. and Culler, D. (2002) Mate: A tiny virtual machine for sensor networks. *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[15] Heidemann, J., Silva, F., Intanagonwiwat, C., Govindan, R., Estrin, D., and Ganesan, D. (2001) Building efficient wireless sensor networks with low-level naming, . New York, NY, USA, October, pp. 146–159. ACM.

[16] Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. (2002) Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Operating Systems Review*, **36**, 131–146.

[17] Arora, A. and Kulkarni, S. (1998) Detectors and correctors: A theory of fault-tolerance components. *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, May.

[18] Arora, A., Demirbas, M., and Kulkarni, S. (2001) Graybox stabilization. *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2001)*, pp. 389–398.

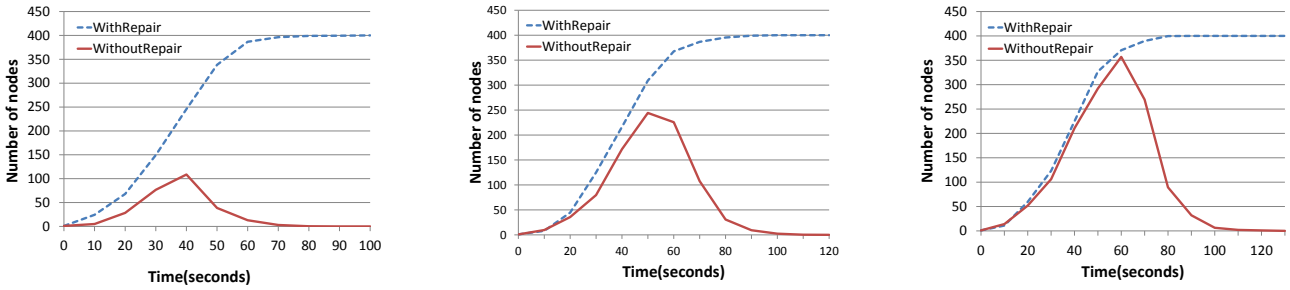


(a) Number of nodes which receive correct code vs Time

(b) Number of nodes which receive correct code vs Time

(c) Number of nodes which receive correct code vs Time

FIGURE 14. Varuna and Varuna ∘ Best-Effort-Repair: 1 faulty node booted at (a) 30 seconds after updated code injection, (b) 45 seconds after updated code injection, (c) 60 seconds after updated code injection.



(a) Number of nodes which receive correct code vs Time

(b) Number of nodes which receive correct code vs Time

(c) Number of nodes which receive correct code vs Time

FIGURE 15. Varuna and Varuna ∘ Best-Effort-Repair: 4 faulty nodes booted at (a) 30 seconds after updated code injection, (b) 45 seconds after updated code injection, (c) 60 seconds after updated code injection.

[19] Levis, P., Lee, N., Welsh, M., and Culler, D. (2003) Tossim: accurate and scalable simulation of entire tinyos applications. *Proceedings of the 1st international conference on Embedded networked sensor systems*, New York, NY, USA SenSys '03, pp. 126–137. ACM.

[20] Stathopoulos, T., Heidemann, J., and Estrin, D. (2003) A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30. UCLA, Center for Embedded Networked Computing.

[21] Reijers, N. and Langendoen, K. (2003) Efficient code distribution in wireless sensor networks. *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, New York, NY, USA WSNA '03, pp. 60–67.

[22] Jeong, J. and Culler, D. (2004) Incremental network programming for wireless sensors. *IEEE Sensor and Ad Hoc Communications and Networks (SECON)*, pp. 25–33.

[23] Gnawali, O., Jang, K., Paek, J., Vieira, M., Govindan, R., B.Greenstein, A.Joki, D.Estrin, and Kohler, E. (2006) The tenet architecture for tiered sensor networks. *SenSys*, pp. 153–166.

[24] Tolle, G. and Culler, D. E. (2005) Design of an application-cooperative management system for wireless sensor networks. *EWSN*, pp. 121–132.

[25] Whitehouse, K., Tolle, G., Taneja, J., Sharp, C., Kim, S., Jeong, J., Hui, J., Dutta, P., and Culler, D. (2006) Marionette: using rpc for interactive development and debugging of wireless embedded networks. *Proceedings of the 5th international conference on Information processing in sensor networks*, New York, NY, USA IPSN '06, pp. 416–423. ACM.

[26] Mottola, L., Picco, G., and Amjad, A. (2008) Figaro: Fine-grained software reconfiguration in wireless sensor networks. *Proceedings of 5th European Conference on Wireless Sensor Networks*.

[27] Pasztor, B. and et al. (2010) Selective reprogramming of mobile sensor networks through social community detection. *Proceedings of European Conference on Wireless Sensor Networks*, pp. 178–193.

[28] Arora, A. and Kulkarni, S. S. (1998) Detectors and correctors: A theory of fault-tolerance components. *Proc. of the* 18^{th} *Int. Conf. on Distributed Computing Systems (ICDCS)*.

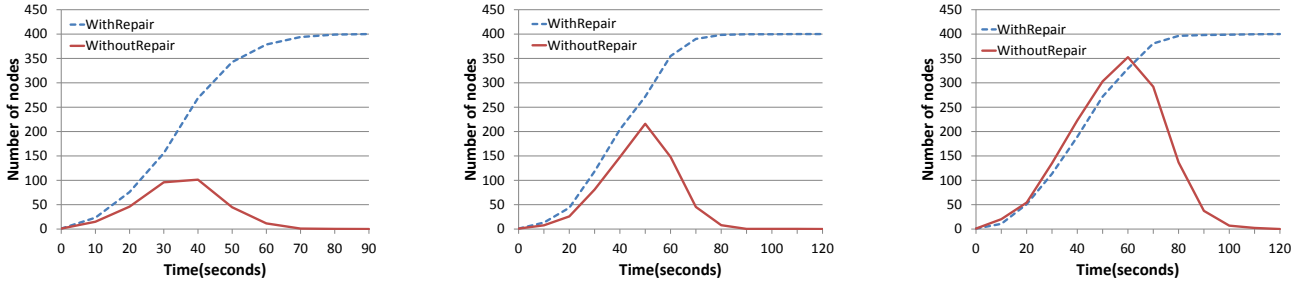
[29] Dolev, S. (2000) *Self-Stabilization*. MIT Press.

[30] Kranakis, E., Krizanc, D., and Pelc, A. (2001) Fault-tolerant broadcasting in radio networks. *Journal of Algorithms*, pp. 47–67.

[31] Koo, C.-Y. (2004) Broadcast in radio networks tolerating byzantine adversarial behavior. *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing* PODC '04, pp. 275–282.

[32] Bhandari, V. and Vaidya, N. H. (2005) On reliable broadcast in a radio network. *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, New York, NY, USA PODC '05, pp. 138–147. ACM.

[33] Koo, C.-Y., Bhandari, V., Katz, J., and Vaidya, N. H. (2006) Reliable broadcast in radio networks: the

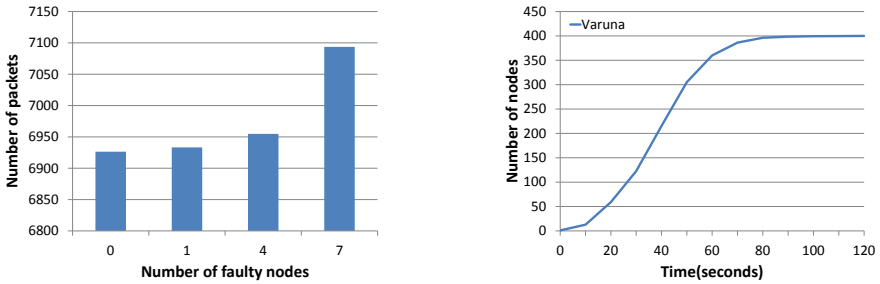


(a) Number of nodes which receive correct code vs Time

(b) Number of nodes which receive correct code vs Time

(c) Number of nodes which receive correct code vs Time

FIGURE 16. Varuna and Varuna ∘ Best-Effort-Repair: 7 faulty nodes booted at (a) 30 seconds after updated code injection, (b) 45 seconds after updated code injection, (c) 60 seconds after updated code injection.



(a) Number of (Repair + ADV) packets sent vs number of corrupted nodes

(b) Completion of Varuna (no transient faults)

FIGURE 17. Varuna ∘ BestEffort-Repair : Network Size: 20 * 20, Nodes Corrupted at Random

bounded collision case. *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, New York, NY, USA PODC '06, pp. 258–264. ACM.

[34] Bertier, M., Kermarrec, A.-M., and Tan, G. (2010) Message-efficient byzantine fault-tolerant broadcast in a multi-hop wireless sensor network. *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, Washington, DC, USA ICDCS '10, pp. 408–417. IEEE Computer Society.

[35] Maurer, A. and Tixeuil, S. (2012) On byzantine broadcast in loosely connected networks. *Proceedings of the 26th international conference on Distributed Computing*, Berlin, Heidelberg DISC'12, pp. 253–266. Springer-Verlag.

[36] Maurer, A. and Tixeuil, S. (2013) On byzantine broadcast in planar graphs. *CoRR*, **abs/1301.2875**.

[37] Dijkstra, E. W. (1974) Self stabilizing systems in spite of distributed control. *Communications of the ACM*, **17**, 643–644.

[38] Burns, J. E., Gouda, M. G., and Miller, R. E. (1993) Stabilization and pseudo-stabilization. *Distributed Computing*, **7**, 35–42.

[39] Alpern, B. and Schneider, F. B. (1985) Defining liveness. *Information Processing Letters*, **21**.

[40] Gärtner, F. C. and Jhumka, A. (2004) Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications. *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, Grenoble, France, September.

[41] Arora, A. et al. (2004) A line in the sand: a wireless sensor network for target detection, classification, and tracking. *Computer Networks*, **46**, 605–634.

[42] Cooprider, N., Archer, W., Eide, E., Gay, D., and Regehr, J. (2007) Efficient memory safety for tinyos. *Proceedings of the 5th international conference on Embedded networked sensor systems*, New York, NY, USA SenSys '07, pp. 205–218. ACM.

[43] Kim, S., Kim, S., and Eom, D. S. (2012) A robust and space-efficient stack management method for wireless sensor network os with scarce hardware resources. *International Journal of Distributed Sensor Networks (IJDSN)*, **2012**.

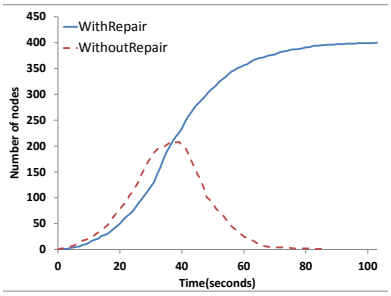
[44] Kumar, R., Singhania, A., Castner, A., Kohler, E., and Srivastava, M. (2007) A system for coarse grained memory protection in tiny embedded processors. *Proceedings of the 44th annual Design Automation Conference*, New York, NY, USA DAC '07, pp. 218–223. ACM.

[45] Demirbas, M. and Balachandran, S. (2007) Robcast: A singlehop reliable broadcast protocol forwireless sensor networks. *27th International Conference on Distributed Computing Systems Workshops (ICDCS 2007 Workshops)*, pp. 54–.

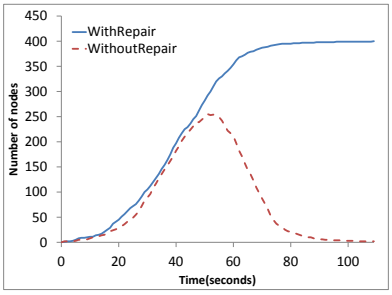
[46] Cinque, M., Cotroneo, D., Martino, C. D., Russo, S., and Testa, A. (2009) Avr-inject: A tool for injecting faults in wireless sensor nodes. *IPDPS*, pp. 1–8. IEEE.

[47] Tinyos. http://docs.tinyos.net.

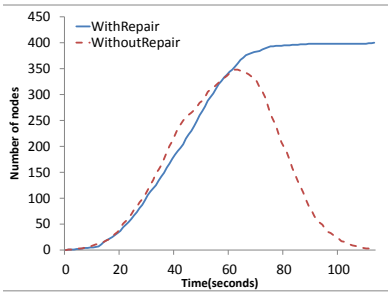
[48] Jhumka, A., Hiller, M., and Suri, N. Approach for designing and assessing detectors for dependable



(a) Number of nodes which receive correct code vs Time

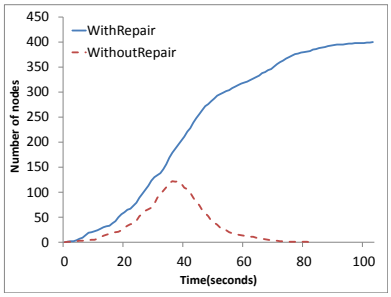


(b) Number of nodes which receive correct code vs Time

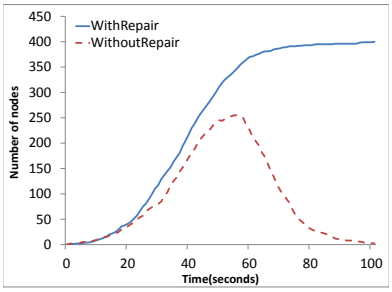


(c) Number of nodes which receive correct code vs Time

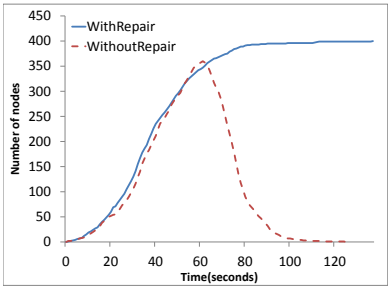
FIGURE 18. Varuna and Varuna ∘ Consistent-Repair: 1 faulty node booted at (a) 30 seconds after updated code injection, (b) 45 seconds after updated code injection, (c) 60 seconds after updated code injection.



(a) Number of nodes which receive correct code vs Time



(b) Number of nodes which receive correct code vs Time

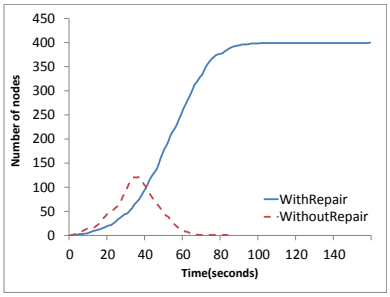


(c) Number of nodes which receive correct code vs Time

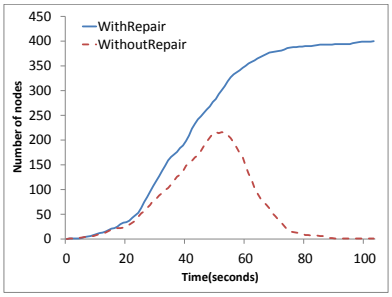
FIGURE 19. Varuna and Varuna ∘ Consistent-Repair: 4 faulty nodes booted at (a) 30 seconds after updated code injection, (b) 45 seconds after updated code injection, (c) 60 seconds after updated code injection.

component-based systems. *HASE*, pp. 69–78.

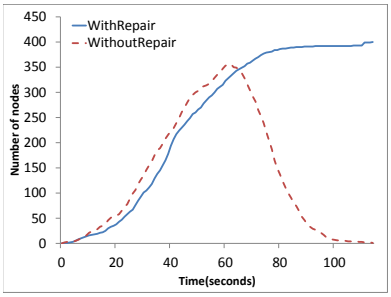
[49] Mottola, L. and Picco, G. (2006) Logical neighborhoods: A programming abstraction for wireless sensor networks. *Proceedings of Distributed Computing in Sensor Systems (DCOSS)*, pp. 150–168.



(a) Number of nodes which receive correct code vs Time



(b) Number of nodes which receive correct code vs Time



(c) Number of nodes which receive correct code vs Time

FIGURE 20. Varuna and Varuna ∘ Consistent-Repair: 7 faulty nodes booted at (a) 30 seconds after updated code injection, (b) 45 seconds after updated code injection, (c) 60 seconds after updated code injection.