Stabilizing Health Monitoring for Wireless Sensor Networks

William Leal, Sandip Bapat, Taewoo Kwon, Pihui Wei, Anish Arora [⋆]

Department of Computer Science and Engineering
The Ohio State University, Columbus, OH 43210
{leal, bapat, kwonta, weip, anish}@cse.ohio-state.edu

Technical Report OSU-CISRC-6/06-TR62

Abstract. Wireless sensor networks (WSNs) comprised of low-cost devices tend to be unreliable, with failures a common phenomenon. Being able to accurately observe the network health status — of nodes of each type and links of each type — is essential to properly configure applications on WSN fabrics and to interpret the information collected from them.

In this paper we study accurate network health monitoring in WSNs. Specifically, we reconsider the well-known problem of message-passing rooted spanning tree construction and its use in PIF (propagation of information with feedback) for the case of a WSN. We present a stabilizing protocol, Chowkidar, that is initiated upon demand; that is, it does not involve ongoing maintenance, and it terminates with accurate results, including detection of failure and restart during the monitoring process. Our protocol is distinguished from others in two important ways. Given the resource constraints of WSNs, it is message-efficient in that it uses only a few messages per node. And it tolerates ongoing node and link failure and node restart, in contrast to requiring that faults stop during convergence.

We have implemented the protocol as part of enabling a network health status service that is tightly integrated with a remotely accessible wireless sensor network testbed, Kansei, at The Ohio State University. We report on experimental results.

1 Introduction

Note: This is an expanded and slightly corrected version of the one published in the proceedings of SSS 2006. Besides including proofs and additional comments, the tree construction protocol is somewhat simpler and more efficient.

⋆ This work was supported in part by NSF grants NSF-NETS/NOSS-0520222 and NSF-HDCCSR-0341703, and by DARPA contract OSU-RF #F33615-01-C-1901.

Wireless sensor networks (WSNs) are inherently unreliable. When WSN nodes, generally built from low cost components, are deployed in large numbers, failures happen fairly often. Monitoring the health of nodes and interfaces of deployed WSNs is thus a core requirement for application managers. Similarly, WSN testbed users have a particular need for accurate information about the health of the testbed. In running an experiment, it is easy to confuse a failed node with a problem in the experiment itself. Experiments use controlled fault injection that simulates node, interface and other failures, so the actual failure of a node or interface will give misleading results. Hence having an accurate view of the network before and after an experiment is important in interpreting results. In addition, the administrator of a testbed needs to know which nodes have failed so that they can be restarted or repaired.

In WSN deployments, monitoring is typically done using exfiltration on multi-hop radio paths that are unreliable and exhibit complex dynamics. Although WSN testbeds generally use reliable, high-speed back channels for experiment configuration and data retrieval that can be exploited to improve the efficiency and accuracy of monitoring, these channels may not always be available due to failures or policy issues such as interference with ongoing experiments. Thus, in either situation, a monitor cannot assume a persistent structure for its status collection, which motivates the need for a tolerant solution.

A simple way to gather health information for any WSN is to use the standard pattern of propagation of information with feedback (PIF). However, for our purposes, the PIF must have certain properties. First, it must give accurate, total results or else indicate that a problem occurred: node and interface failures can (and do) happen at any time, including during the running of the protocol itself; this can result in dramatically inaccurate results if, for example, a tree is formed but a node close to the root fails before feedback completes. Even if the node does not fail, its wireless link to its parent could become unreliable due to complex WSN link dynamics or channel interference, resulting in loss of information from that node. These scenarios could result in a partial report where the entire subtree dominated by that node is regarded as inaccessible when in fact an alternative reliable path to those nodes might exist. In such cases, partial PIFs would yield information that is, from the perspective of the users, potentially worse than no information.

Second, the protocol should be fast. We wish to run it between experiments on a testbed so it should not take long. Third, the protocol must be frugal in its use of communication resources since it shares the network with other user programs and might run on battery-based nodes. Fourth, for the same reason, the protocol should run on demand and should terminate, not using resources when network information is not being collected. This precludes approaches that perform automatic tree repair when nodes or links fail or restart. Fifth, the protocol should deliver monitoring results to a central location so that users and administrators can assess them. Sixth, since testbeds involve a variety of different kinds of networks, it should not assume any particular kind of network, such as TCP/IP; rather, it should work on any network or combination of networks.

Since WSNs can be heterogeneous there can be many paths from the root to a node. A path that uses Ethernet will usually be more reliable than one that uses radio, for instance. Hence we prefer least-cost paths, where the cost of a link is a reflection of its reliability or other factors such as bandwidth. Thus, for instance, we can avoid low bandwidth multi-hop radio paths that could have higher bit error rates or be more prone to message interference unless they are the only way to reach certain nodes.

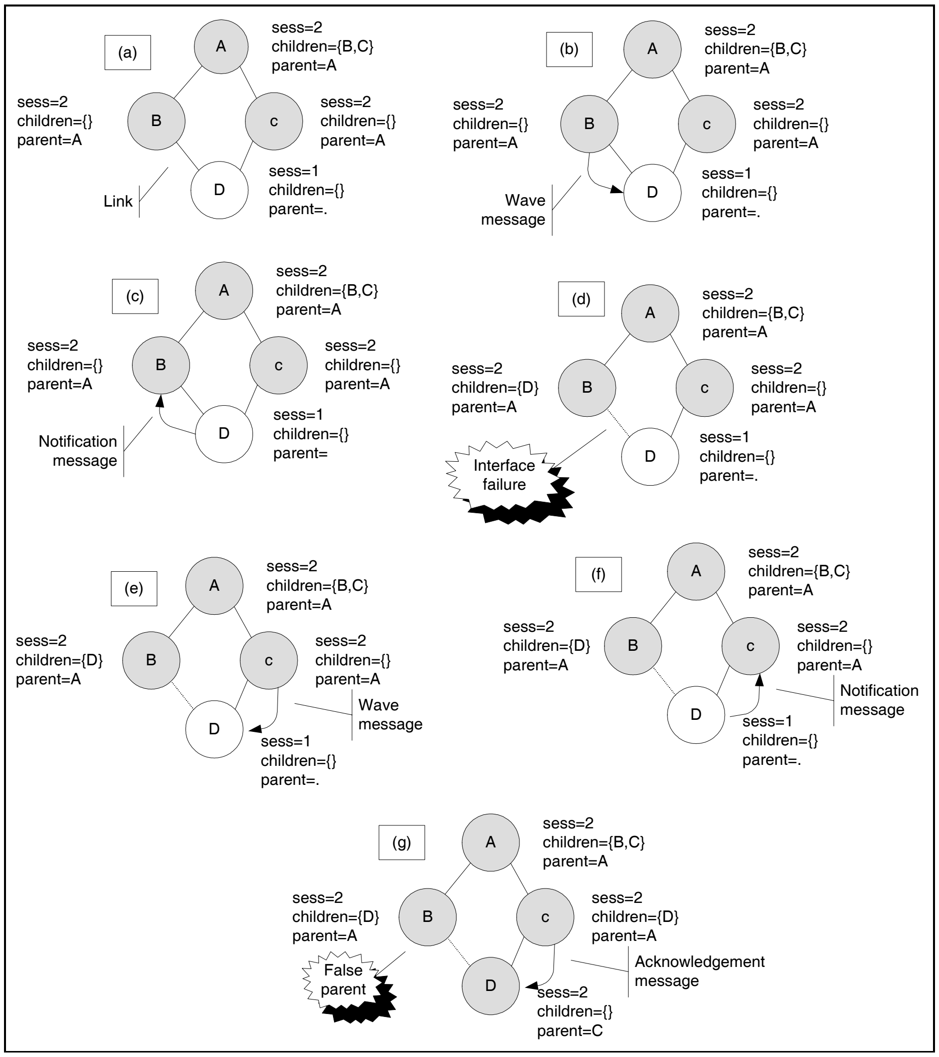

(a)
A
sess=2 children={B,C} parent=A
B
sess=2 children={} parent=A
c
sess=2 children={} parent=A
D
sess=1 children={} parent=.
Link
(b)
Wave message
(c)
Notification message
sess=1 children={} parent=
(d)
sess=2 children={D} parent=A
Interface failure
(e)
Wave message
(f)
Notification message
(g)
sess=2 children={D} parent=A
False parent
Acknowledgement message
sess=2 children={} parent=C

Figure 1: Tolerating Failures During Acknowledgement: Unresponsive Children

Designing for heterogeneity means that our approach is easily extended to new kinds of devices and networks. We are, for example, in the process of incorporating new motes that use new interfaces into our implementation.

Outline of our Chowkidar protocol.[1] Existing PIF-style approaches do not satisfy the above requirements. They either assume that faults have stopped or else they run continuously, often using substantial resources when a fault occurs. By making judicious use of time, we present a simple, efficient protocol that produces in one shot a tree structure such that, in the presence of continuous faults, a subsequent PIF will either succeed or will indicate that a fault has rendered the tree defective. In this latter case, the tree can be rebuilt and a new PIF run.

Faults can cause a link or node to fail and a node to restart; in this latter case, local variables are reset and a restart message is broadcast that can be forwarded to the root[2]. Faults do not affect other variables.

Key to our approach is the tree-building protocol that includes a handshake between a node and its potential parent. When X receives a wave broadcast from Y with higher session number, it asks Y to become its parent. Y records X as a child and sends an acknowledgement. If the acknowledgement fails then X does not adopt Y as parent. In a subsequent PIF, Y expects to hear from X and when it doesn't, it knows there is a tree fault that requires a rebuild. We accomplish formation of least-cost paths by phasing the delivery of the wave messages: on links with lower cost, the messages are delivered earlier than on links with higher cost. A node that is connected to a neighbor on multiple networks will receive the message on the preferred network first. If faults do not occur during handshaking then the tree will be least-cost.

If node or link failures do not happen during the tree formation, the result is a tree with bidirectional edges: each child knows its parent and each parent knows its children; when a PIF is run on the tree, each parent waits on its children to report before it reports to its own parent and, if it fails to hear from a child in a timely fashion, it initiates a failure message to the root. As noted, the handshake process lets us handle failures that occur during the acknowledgement sequence. If a node fails to receive the acknowledgement from the proposed parent then it does not join the tree but waits for another wave message from another neighbor. As shown in Fig. 1, this means that it is possible for a parent to incorrectly claim a node as child. In the figure, node A is the root and nodes B and C have joined the tree. In Fig. 1(b) to Fig. 1(d), a wave message reaches node D and it notifies B that it proposes adopting it as parent, so B includes D in its child set. However, a failure just after that causes the link to fail, so D does not join the tree. In Fig. 1(e) to Fig. 1(g), a wave message arrives from C and D joins with C as parent. Note now that both B and C claim D as a child. In a subsequent PIF, D will give feedback to C but not to B. When B fails to hear from D, it will respond with an error message to the root.

Fig. 1 shows how an *unresponsive child* can be formed as the result of a link failure during tree formation. In a properly formed tree, a child can later become unresponsive if it or the link to its parent fails.

[1] Chowkidar in the Hindi language stands for a watchman.

[2] Links can restart silently; detecting a restart efficiently is an issue we do not deal with here.

When tree formation or PIF is complete, the protocol is quiescent, so there is no ongoing message traffic unless a node restarts. In the absence of failures, a total of three messages per node are required for tree formation: one for the wave, two for the parent acknowledgement. For a PIF, two messages per node are required: one to propagate the token and one to propagate the feedback. If failures occur during the parent acknowledgement process, additional messages are required as a node attempts to confirm with subsequent potential parents. However, this occurs only if a failure happens after the wave message is sent but before the acknowledgement arrives.

Contributions. Our principal contribution is the message-passing terminating spanning tree protocol of Chowkidar that tolerates on-going faults. The structure is produced in such a way that a subsequent PIF will either succeed or will report that the structure is not a spanning tree.

Further, we analyze the protocol in the context of a formal network model, and offer experimental validation of the protocol performance via an implementation on a heterogenous WSN testbed, Kansei, comprising hundreds of Motes (of multiple types, specifically XSMs and TMoteSkys), Stargates, and PCs. The latter is necessary in part for validating that the performance in a real network (with associated complex dynamics) is consistent with that predicted by the analysis. It is also necessary for enabling a health monitoring service that is a crucial and tightly integrated component of Kansei.

Organization of the paper. The rest of the paper is organized is follows. We discuss related work in Section 2. The Chowkidar tree and PIF protocols are in Section 3. We give experimental results in Section 4 and present concluding remarks in Section 5. Appendices contain the correctness proof for the tree protocol, a photo of the Kansei testbed, and screenshots for an implementation of Chowkidar.

2 Related Work

Efficient PIF with ongoing faults. The notion of a self-stabilizing PIF has been well-studied in distributed computing as it is an enabler for many other tasks such as distributed reset, global snapshot, termination detection, and others; see [1] for an overview of non-stabilizing, self-stabilizing and snap-stabilizing PIF protocols; in fact, our protocol is snap-stabilizing, completing in zero rounds. However, existing protocols that terminate do not tolerate ongoing faults, and those that do are not terminating.

[2] presents a PIF in the form of distributed reset. It is tolerant to ongoing faults, returning either a correct completion or an error indicating a tree failure. However, the protocol is not terminating since the tree structure is checked periodically and repair occurs when a problem is detected. [3] and [4] give terminating tree construction protocols but they make the standard stabilization assumption that faults have stopped.

Network health monitoring. A variety of monitoring facilities have been developed for testbeds and for deployed WSNs, but all those we are aware of fall short of our needs. Experiments are assumed to run on homogeneous devices; existing support tools do not handle the network heterogeneity of testbeds. Tools do not distinguish between the health of a node and the health of its links. For some, the reliability is too low to be useful and for others, there is a dependency on the communication structure of the application.

Traditional networks such as the Internet use standard protocols such as SNMP [5] for monitoring network devices and identifying faults. However, SNMP assumes the IP routing layer in its operation and is therefore dependent on the fault-tolerance of IP to be able to reach the monitored devices; this is not sufficient for a WSN with non-IP networks such as mote radio. Other monitors like Sympathy [6] only handle radio networks and do not admit heterogeneity.

Similarly, Motelab [7], Tutornet [8] and Orbit [9] provide users with a ping-based status for each device, indicating whether it is reachable or not. However, simply detecting that a device is unresponsive on a given network is not sufficient since it does not support heterogeneous networks and does not distinguish network and link faults.

The Sensor Network Management System (SNMS) [10] supports monitoring of WSNs by providing its own network stack that includes routing. SNMS allows administrators to remotely query network devices and learn their status. However, SNMS does not deal with heterogeneous networks, and studies such as [11] show that reliability of SNMS does not suffice to provide accurate fault status.

Sympathy [6] is designed for fault detection at a central base station in a data collection application in which nodes periodically send data to the base. Sympathy thus exploits knowledge of a specific application's traffic pattern to define certain fault metrics. A similar approach is used in [12] where the fault management system exploits the continuous data traffic flow in the network to piggy-back health information and uses route update messages to trace failed nodes. The dependency on an application makes these inappropriate for our purposes since monitoring is conducted only when an application is running.

3 Tree Construction and PIF for WSN Health Monitoring

In this section, we present our protocol for tree construction and for PIF. A client using the protocol would first execute the tree construction and when that terminates, execute the PIF. If the PIF returns an error then the tree must be rebuilt and the PIF rerun. If not, the tree that has been formed can be used again for the next PIF. For efficiency, the tree construction can be combined with an initial feedback from the leaves.

Communication Model. For purposes of design and analysis, we assume that links are bidirectional and reliable. Both assumptions need to be justified. In WSNs, unidirectional links may occur since a node might hear from a distant neighbor but messages sent in the other direction, for a variety of reasons, may not be received. In our tree construction protocol, a node must both send and receive on a selected link before it joins the tree, so a unidirectional link would be ignored or regarded as failed, and another parent/link candidate, if any, would be attempted. In any case, unidirectional links can be handled by estimating link quality or using topological information so that a node only accepts messages from near neighbors; however, link quality estimation usually requires exchanging several messages, which our protocol does not support, so estimation would have to be provided by a separate service.

To simplify our model we assume that links are reliable if they have not failed due to an interface fault. In reality, radio-based links tend to be unreliable due to contention for the channel. The problem can be mitigated by using low power and randomized transmission times, but the results of a PIF collection will be at best probabilistic. Alternatively, a TDMA-based scheme can be used to give deterministic communication, but TDMA itself comes with an overhead cost unless it is already provided for some other purpose. In Section 4 we describe a simple approach that gives adequately reliable radio communications without continuous overhead.

Fault Model. We assume a base station as a fixed root that is not affected by faults. This assumption is justified in our application since a base station is relatively reliable. Failure of the base station is obvious to users, and it has to be repaired or replaced for correct network operation. The session number that is used to initiate tree construction is stored by the base station and is assumed to be reliable. For convenience we assume that session numbers are unbounded, but since the base is reliable, an integer with a sufficiently large bound suffices.

Faults can cause node or link to failure or node restart; as mentioned earlier, we do not consider link restart, which is postponed to future work. Node restart is clean in that the session number and other protocol control variables are reinitialized; it is detectable by the node so that when it happens, it broadcasts a “restart” message that is passed to the root.

Notation and Semantics. We use guarded commands with interleaving semantics for specifying the protocol. Parameters `j` and `k` range over nodes; `i` ranges over links. Among commands with true guards, one is chosen and is atomically executed. Communication actions are synchronous: after a node sends a message to `X`, `X` executes the corresponding receive before any other action of `X` and before any other node sends to `X`. Environment actions can interleave send-receive pairs, might or might not be executed when enabled, and execute at most finitely often.

We use Gouda’s AP notation for timeouts, which are represented by global state predicates. In the case of the child-parent acknowledgement sequence during tree construction (T6 of Fig. 2), for instance, the timeout can be implemented by setting a time to wait for an acknowledgement after notification has been sent. Termination of each protocol can be given via a timeout predicate that negates the conjunction of the guards.

Timing Assumptions. We use delayed delivery of wave messages to construct a least-cost tree, implemented by an underlying broadcast service that delays actual broadcast until a specified time. In our implementation we have the following delays: Ethernet, 0s; serial link between Stargates and XSMs, 0s; XSM radio, 5s, where higher delay corresponds to higher link cost. From the base station to a given XSM X_0 there can be many potential paths, including B-E-SG_0-S-X_0 and B-E-SG_1-S-X_1-R-X_0, where B is base, SG is a Stargate, X is an XSM, E is Ethernet, S is serial link and R is radio. The first path costs 0s and the second 5s, so the first is preferred. Delayed delivery means that the wave via the serial link arrives at X_0 before the wave via the radio, so SG_0 will be selected as X_0's parent.

We assume that internal processing in nodes takes zero time and that links have associated timing values, including the delay enforced by the broadcast primitive. This lets us associate an upper bound on the time to wait before concluding that a timeout is satisfied.

Constants, Variables and Messages. We describe the constants, node variables, environment variables and message types. **Constants.** `lktype` is a constant set of link types for a given node corresponding to interface types. `id` is the identifier for a node; we assume that nodes have unique identities. **Node Vars for Both Protocols.** `sn` is an unbounded integer session number, initially 0. **Node Vars for Tree Construction.** `parent` is the (node id, interface) pair for a parent, initially (-1,-1) for unassigned. `child` is a set of of (node id, interface) pairs of children, initially $\emptyset$. `busy` ensures only one handshake at a time, initially false. `tk, ti, tsn`, initially -1, -1, 0, resp., store parent candidate information during handshake. `newtree` is true if a new tree should be formed, initially true. **Node Vars for PIF.** `pphase` is a Boolean that is true during the propagation phase, initially false. `pchldcnt` is a bounded integer that counts the number of children heard from during feedback. **Environment Vars.** `up` is an array of Booleans indexed by node ids that indicates whether the node is up or not. `link` is an array of Booleans indexed by node id pairs and link type that indicates whether the link is up or not; this is symmetric wrt nodes. **Message Types.** Each message begins with a message type. `'wave'` is for tree wave messages. `'notify'` is for a child to notify a potential parent in tree construction. `'ack'` is for the parent to acknowledge the child in tree construction. `'token'` is the propagation message for PIF. `'fb'` is the feedback message for PIF. `'err'` is the error message for the PIF when the structure is found to not be a tree. `'restart'` is issued by a node that restarts.

The tree construction protocol uses several queue functions. `enqueue(k,i,msn)` appends the triple `(k,i,msn)` to the end of the queue which is stored in variable `q`. `queuesn()` returns the the queue's session number (the protocol ensures that all entries have the same session number), zero if the queue is empty. `hdqueue()` returns the first entry in the queue while `dequeue()` does the same thing but also removes the entry.

Tree Construction Protocol. The Chowkidar protocol for tree construction is given in Fig. 2.

- Action T1 initiates a new wave in response to the need for a new tree. Note that the broadcast service ensures that wave messages are delivered in order of link cost.
- Action T2 receives the wave message from a neighbor over some interface and enqueues it; the queue stores other alternatives in case faults prevent the sender of the first wave message from being adopted as a parent.

  Given that a wave message with session number s is in the queue, we are only interested in new wave messages with session number s or higher. In case we hear a higher wave message, we discard the contents of the queue in favor of the higher number.

  Hence the session numbers of enqueued wave messages will always be the same and will always be the highest received so far. This mechanism is in place to ensure that the current session number, if received, supersedes all lower ones. This is not necessary for safety but is necessary for liveness.
- Action T3 removes a wave message if the head of the queue contains a sequence number larger than the current session of the node and hence larger than the session of any other handshake that may be in process. Thus handshakes that may be in process with earlier session numbers might be abandoned; the filter in T5 causes those old responses to be ignored.

  A notify message that is sent to a node in the current session might not be received if the destination node is down or if the link is down. In the former case, the invariant is violated; in the latter, the timeout of T6 will detect the problem and set the node to try for another parent candidate, if available.
- Action T4 receives a notify message and, if the message's session number is the same as the node's, responds with an acknowledgement, adding the sender to its child set.

  The check for session number is not strictly required for soundness. It aborts handshakes from old sessions and so improves efficiency.
- Action T5 receives an acknowledgement from k, and, if the message's sequence number is the same as that of the last wave, makes k its parent, and propagates the wave.

  Consider the possibility that a node might be involved in a simultaneous handshake with two or more other nodes. This can happen when x has started a handshake with y1 and then a new session begins, x abandons the handshake with y1, and begins a handshake with y2. In this case, either acknowledgement could arrive before the other. Suppose the acknowledgement arrives from y1 first. Then x makes y2 its parent and propagates the wave. Later when the acknowledgement from y2 arrives, it again makes y2 its parent and propagates the wave. The same thing happens if the acknowledgements arrive in the reverse order. This does not violate the tree structure but does result in needless wave messages being propagated. The test on the sequence number eliminates this duplication.

```
T1  (id=Base∧newtree) → // begin new tree
    sn:=sn+1; tsn:=sn; parent:=(-1,-1); newtree:=false;
    ∀pi : (pi ∈ lktype) : broadcast (‘wave’,sn) on pi
      atTime tdeliv[pi];
    child:=∅;
T2  rcv (‘wave’,msn) from k on i → // enqueue waves
    if (msn>queuesn()) q:=(k,i,msn);
    elseif (msn=queuesn()) enqueue(k,i,msn);
    fi
T3  hdqueue()=(xk,xi,msn) ∧ msn>tsn → // begin handshake
    (tk,ti,tsn):=dequeue();
    send (‘notify’,tsn) to tk on ti;
T4  rcv (‘notify’,msn) from k on i → // acknowledge
    if(msn=sn)
      send (‘ack’,sn) to k on i;
      child:=child ∪ (k,i);
    fi
T5  rcv (‘ack’,msn) from k on i → // join parent & echo wave
    if (msn=tsn)
      sn:=tsn; parent:=(tk,ti);
      ∀pi : (pi ∈ lktype) : broadcast (‘wave’,sn) on pi
        atTime tdeliv[pi];
      child:=∅; fi
T6  timeout (tsn≠sn ∧ (¬upn.tk ∨ ¬upl.id.tk.ti)) →
    tsn:=sn;
```

Figure 2: Tree Construction Protocol

A simultaneous handshake can also happen when x has started a handshake with y1 in the current session, then x fails and restarts, and starts a handshake with y2. If this happens, then both y1 and y2 will claim x as a child, yielding a false child that will be detected by the pif.

Node and link faults may occur concurrently with the protocol. If a node or link fails before the wave reaches it then this is the same as failing before the protocol begins. If a node or interface used in the tree fails after the wave has passed, this is the same as failing after the protocol terminates and will be detected by a subsequent PIF since that node will fail to respond to its parent.

Now consider failures concurrent with the wave boundary. Suppose node `Y` broadcasts via action T1 or T5, but before delivery, neighbor `X` or the link fails. This is the same as failure before the protocol begins and `X` will not be included in the tree. Suppose `X` sends notification to `Y` via T3 but `Y` or the link fails before receipt. This is the same as failing before the protocol starts. Suppose `Y` receives a notification from `X` in T4 but the link fails before `X` receives the acknowledgement. Then `Y` adopts `X` as a child but `X` does not learn this; T6 will cause a timeout and `X` will solicit a new parent. Since `Y` has falsely recorded `X` as a child, in a subsequent PIF will detect that X does not respond to Y. Similarly, if Y receives a notification from X but X fails before receiving the acknowledgement, X will not be responsive in the subsequent PIF.

```
P1  (id=Base) ∧ (start new PIF) ∧ ¬newtree → //new PIF
    sn:=sn+1;
    ∀pk,pi : (pk,pi) ∈ child : send (‘token’,sn)
      to pk on pi;
    pphase:=true; pchldcnt:=0;
P2  rcv (‘token’,psn) from k on i → //propagate or respond
    if (psn > sn)
      sn:=psn;
      if (child≠ ∅)
        ∀pk,pi : (pk,pi) ∈ child : send (‘token’,sn)
          to pk on pi;
        pphase:=true; pchldcnt:=0;
      else
        send (‘fb’,sn) to parent.node on parent.link;
        pphase:=false; fi fi
P3  rcv (fb,psn) from k on i → //forward responses
    if (psn=sn)
      pchldcnt:=pchldcnt+1;
      if (pchldcnt=|child|)
        if (id≠Base) send (‘fb’,sn) to parent.node
            on parent.link; fi
        pphase:=false; fi fi
P4  timeout pphase ∧ (k,i)∈child ∧ (¬upn.k ∨ ¬link.id.k.i ∨
        parent.k≠(j,i)) → //timeout unresponsive child
    if (id=Base) newtree:=true;
    else send (‘err’) to parent.node on parent.link; fi
    pphase:=false;
P5  rcv (‘err’) from k on i → //send error to Base
    if (id=Base) newtree:=true;
    else send (‘err’) to parent.node on parent.link; fi
```

Figure 3: PIF Protocol

The protocol tolerates simultaneous sessions. In the absence of faults, the protocol will form a spanning tree over the highest session number among non-tree nodes that were up when the wave was propagated by a neighbor. In the presence of faults that affect the tree structure, either the fault happens before the wave or else the structure contains nodes with nonresponsive children. As we have seen, a node listed as a child can be nonresponsive if it or its tree interface is down, or if it has identified a different parent. In this case, a subsequent PIF will detect the fact and initiate a new tree construction.

In the protocol, wave messages are delivered in order according to cost of the links. Hence the spanning tree formed is least-cost unless while handshaking with node Y, the link fails before X's notification is sent, causing X to attempt another parent which, if available, might result in a path that is not least-cost. The only active nodes are those that are at the wave boundary. Timeouts guarantee that wave progresses until it is extinguished, so the protocol terminates. Since we have assumptions about delivery and link times, we can calculate an upper bound on the time required to terminate. Proofs are in the Appendix.

PIF Protocol. The Chowkidar PIF protocol is shown in Fig. 3. For WSN health monitoring, PIF should return appropriate health information or assessment of other predicates but this is an orthogonal issue.

P1 blocks if a tree fault has been detected and should be initiated only when the tree protocol has terminated. A token is initiated on the tree in the form of a message with a new sequence number. In P2, interior nodes forward the token to their children while leaf nodes begin the feedback response. In P3, a node that receives feedback from a child increments a counter; when all children have responded, it sends its feedback response to its parent. In P4, a node waiting on a child's response times out if it is unresponsive. This creates a message that is propagated up the tree by P5.

Implementation of timeout P4 can be based on the longest possible path in the network. An initial PIF can refine the timeouts based on a node's distance from its leaves to get tighter values.

A PIF execution terminates within some bounded time and, if the structure from the root is a spanning tree, then it completes with a report of success and otherwise with a report of failure. If a restart message is received by a node before it has completed feedback then the tree is not spanning; by our synchronous communication assumptions, the restart message will reach the root before the feedback message, triggering the creation of a new tree.

```
E1  id≠Base ∧ up → up:=false; //fail a node
E2  link.id.k.i → link.id.k.i,link.k.id.i:=false,false;
        //fail a link
E3  ¬up → //restart a failed node
    up:=true;
    (reset all variables)
    ∀pi : (pi∈lktype) : broadcast (‘restart’) on pi
        delivery now;
R1  rcv (‘restart’) from k on i → //send restart msg up tree
    if (id=Base) newtree:=true;
    else send (‘restart’) to parent.node on parent.link; fi
```

Figure 4: Environment and Restart Actions

Environment Actions. The environment-related actions are shown in Fig. 4. E1 causes a node to fail. E2 causes a link to fail. E3 causes a node to restart; a restart message is broadcast on all available interfaces.

If R1 receives a restart message, it resets the tree construction busy flag in case the restarted node had been involved in a handshake. To ensure correctness of collection, the message is forwarded towards the root so that a new tree can be formed. Consider the following cases. Suppose the restart message is received by a node with a tree path to the root that stays up sufficiently long. Then the message will arrive at the root. Suppose the receiving node's tree path has a failed node or interface closer to the root. Then a subsequent PIF will detect the problem and when the ensuing tree is formed, the restarted node will be included. Suppose the node is partitioned from tree nodes but there is a newly-restarted neighbor that is not. Then its reset message will trigger a tree rebuild that includes both. Otherwise the node is not reachable by any path of up nodes/links and would not be included the spanning tree even if it were rebuilt.

4 Experimental Results

At Ohio State, we have developed Kansei, a rich hardware-software platform for high-fidelity WSN experimentation, testing, and validation [13]. Its hardware platform couples a generic platform array with multiple domain sensing and communication arrays. The generic platform array consists of a stationary component that can be operated in real-time and via the Internet. It has several hundred static nodes that reside on an off-floor deck composed of 35 bench modules. The two main sensor node platforms in the stationary array, mounted below the deck, are XSM and TMoteSky motes and Stargates. Each XSM has a 4 MHz CPU, 4KB RAM and a low-power single channel 38.4kbps radio. Its sensors and actuators include photocell, PIR, temperature, magnetometer, microphone, GPS, and buzzers. Each TMoteSky is an 802.15.4 compliant device with a 250kbps radio, 8 MHz CPU and 10KB RAM. Each Stargate has a 400MHz Intel PXA255 CPU and a daughter-card with interfaces to the XSM and the TMoteSky and various other interfaces such as RS-232, Ethernet, USB and 802.11(b). The generic platform array also includes mobile sensor nodes that move on the deck above the static array.

We implemented and experimented with Chowkidar for stabilizing tree construction and PIF in Kansei. The implementation for this heterogenous network spans a PC-based Kansei server, Stargates running Linux and motes running TinyOS [14]. It uses Ethernet, 802.11b wireless, mote radio and Stargate-mote serial links for tree construction and data collection. Recall that our protocol uses different delays on different links to construct a least-cost path tree. According to Kansei policies, based on link characteristics such as reliability, available bandwidth, potential interference, etc., we assign a link delay of 0s on Ethernet and Stargate-XSM serial links, which are reliable, have high bandwidth and low interference effects; a 5s delay on XSM radio links, which have less bandwidth and more interference and a 10s delay on 802.11b links. 802.11b links are highest cost since we have only a single channel available and Chowkidar could interfere with concurrent experiments.

Our analysis and proofs assume that links are reliable and bidirectional. However, in reality, broadcast links such as wireless radio and even Ethernet suffer from transient message losses. Indeed, a naive implementation of Chowkidar where child-parent handshakes were initiated immediately upon receiving a wave message led to message implosion on these shared channels and loss of messages in the network due to contention. This network unreliability due to concurrent message transmission by all nodes was in fact so high that it affected not only the performance but also the correctness of our protocol since it resulted in an incomplete tree being constructed in several runs.

To avoid message implosion on shared channels, our implementation introduced a simple application-level backoff mechanism. Coarse-grained tuning of backoffs enabled us to obtain correct performance for our protocol implementation in that a tree spanning all correct nodes was constructed, albeit with increased execution time.

Percentage of failed Stargates	0%	8%	20%	40%
Average time for tree construction	1.2s	8.7s	9.9s	10.5s
Percentage of failed runs	0%	0%	10%	30%

Table 1: Effect of faults on protocol performance
for a 25 node network using a 2.5s backoff.

Table 1 shows the results of the first series of experiments performed on a 25 node network using a 2.5 second backoff on wireless links for congestion avoidance. We first ran our algorithm without introducing any failures. As expected, our algorithm always constructed a least-cost spanning tree using only Ethernet and serial links in very short time. We then injected failures by randomly stopping multiple Stargate nodes, which forced more and more wireless links to be used in tree construction. As the data shows, as the number of injected faults increases, not only does the average tree construction time increase, but in the worst case shown here, where 40% of the Stargates were failed, even correctness was affected in some runs due to excessive message loss. Fortunately, the degradation in performance is gradual and sublinear, so we can select a suitable backoff period based on the expected worst case failures in the network.

Number of network nodes	25	25	50	50
Maximum backoff for radio	2.5s	5s	5s	10s
Percentage of failed runs	30%	0%	25%	0%

Table 2: Linear scaling of backoff with network size for 40% Stargate failures.

We validate this assertion in Table 2 in which we measure the minimum backoff period at which 100% of the runs are successful even in the worst failure case considered in Table 1, i.e. 40% failed Stargates for networks with different sizes. As seen from the data, using a 5 second backoff guarantees correct execution when up to 10 out of 25 (40%) nodes fail; however the same backoff does not guarantee correct execution if the network size is doubled to 50 nodes with the same failure rate. Nevertheless, as expected, with a linear scaling of the backoff period to 10 seconds, we observe correct protocol execution.

We thus conclude from our experiments that in WSNs, unreliability of message transmission affects both protocol correctness and performance and hence should be given careful consideration. However, as demonstrated by our experiments, using a simple backoff mechanism is sufficient to achieve correctness at the expense of increased completion time. Our assumption of reliable, bidirectional links can therefore be reasonably realized even in real network deployments.

An alternate means of obtaining reliable links and guaranteeing protocol correctness is to instrument reliability at the messaging layer either through MAC-level retransmissions or by replacing the CSMA based wireless MAC protocols with TDMA. In future work, we intend to evaluate the cost-benefit tradeoffs of these approaches by comparing their performance with that of the best performance that can be obtained using application-level tuning.

5 Conclusions

We have developed a WSN monitoring protocol that is extensible, has good energy dynamics, and gives accurate results in the presence of ongoing faults. We have given a theoretical model of the protocol and evaluated it against realistic assumptions, so our validation is based not only on analysis but also on experimentation with the Kansei WSN testbed. In the protocol, predicates about network control state can be evaluated locally and consolidated in-network to reduce message traffic; and further evaluation can take place at the base station.

For future work, we will focus on a variety of issues. At present, health monitoring in Kansei is an independent service. Users can view the network health status, but it is not used by Kansei's scheduler. We plan to integrate monitoring so that experiments are run only on nodes known to be good.

Monitoring is presently done only on nodes not running any application. Monitoring concurrently with a running application is desirable, but one would have to deal with the issue of interference between the application and the monitoring. This could be in part a policy issue for a network—an application may request that monitoring of a certain sort not be run concurrently, say on motes—and partly a research issue in case there is a way to exploit the semantics of an application for monitoring while still offering correctness guarantees.

A node interface has two parts, a transmitter and a receiver. Evaluating the receiver locally is easy; but a neighbor is needed to evaluate the transmitter. Broadcasts may be heard by many neighbors and if they all report it, there is excessive redundancy. Hence, we will study how to compress such information. There may additionally be other predicates that involve a node's neighbors, but those neighbors could be in different subtrees, so the structure of the spanning tree could work against us. We will study ways to ameliorate this problem.

Link quality is an important predicate for monitoring. This requires the exchange of several predicates before an evaluation can be made. Over time, if experiments or monitoring are run sufficiently often, link quality status can be assessed; or an estimation algorithm can be run periodically. This can provide information to users, and can also feedback into the monitoring protocol itself: a node can dynamically adjust link cost to build a better tree.

We are interested in the quality of sensors on nodes. The problem is difficult for several reasons. First, sometimes ground truth is not available, so there is no absolute reference point for evaluating the readings [15]. Second, an understanding of the physical model is critical, especially when comparing the readings from nearby sensors. Third, an understanding of the effect of hardware and other environmental faults is important.

As scale increases, the issue of bidirectional link reliability becomes increasingly critical. We need to evaluate whether interference-detection CSMA approaches combined with appropriate timings give us sufficiently accurate results or whether a deterministic scheme such as TDMA is necessary.

DAGs have been proposed as more suited to sensor networks than trees [16], and we plan to investigate this option.

References

1. A. Cournier, A. K. Datta, F. Petit, and V. Villain. Enabling snap-stabilization. In *Proceedings of ICDCS 2003*, 2003.
2. S. Kulkarni and A. Arora. Multitolerance in distributed reset. *Chicago Journal of Computer Science*, 4, 1998.
3. B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction (extended abstract). In *Proceedings of 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.
4. A. Cournier, F. S. Devismes, and V. Villain. Snap-stabilizing PIF and useless computations. In *Proceedings of 12th International Conference on Parallel and Distributed Systems - Volume 1 (ICPADS'06)*, pages 39–48, 2006.
5. IETF. RFC 1157. `www.ietf.org/rfc/rfc1157.txt`.
6. N. Ramanathan et al. Sympathy for the sensor network debugger. In *SenSys '05: 3rd Intl. Conf. on Embedded networked sensor systems*, pages 255–267, 2005.
7. G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: A Wireless Sensor Network Testbed. In *4th Intl Conf on Information Processing in Sensor Networks*, 2005.
8. Embedded Networks Laboratory, USC. Tutornet: A Tiered Wireless Sensor Network Testbed. `http://enl.usc.edu/projects/tutornet/index.html`.
9. D. Raychaudhuri et al. Overview of the ORBIT Radio Grid Testbed for Evaluation of Next-Generation Wireless Network Protocols. In *IEEE Wireless Communications and Networking Conference (WCNC)*, 2005.
10. G. Tolle and D. Culler. Design of an Application-Cooperative Management System for Wireless Sensor Networks. In *Proceedings of the EWSN'04*, 2004.
11. S. Bapat, V. Kulathumani, and A. Arora. Analyzing the Yield of ExScal, a Large-Scale Wireless Sensor Network Experiment. In *13th IEEE Intl. Conf. on Network Protocols (ICNP)*, pages 53–62, 2005.
12. J. Staddon, D. Balfanz, and G. Durfee. Efficient tracing of failed nodes in sensor networks. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 122–130, 2002.

13. A. Arora, E. Ertin, R. Ramnath, M. Nesterenko, and W. Leal. Kansei: A high-fidelity sensing testbed. *IEEE Internet Computing*, 10(2):35–47, March/April 2006.
14. J. Hill et al. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.
15. N. Ramanathan et al. Rapid deployment with confidence: Calibration and fault detection in environmental sensor networks. Technical Report CENS 62, Center for Embedded Network Systems, UCLA, 2006.
16. S. Nath et al. Synopsis diffusion for robust aggregation in sensor networks. In *2nd Intl. Conf. on Embedded Networked Sensor Systems*, pages 205–262, 2004.

Appendices

A Proof of Correctness for Tree Construction

A.1 Soundness of Tree Construction

Proof Outline. The proof for the tree construction protocol is in three parts: correctness, completeness and least-cost.

For soundness, we assume asynchronous operation and give an invariant-based proof that the result is either a tree with bidirectional edges or else there is a node with a child that is unresponsive.

For liveness, we give a variant-based proof that not only proves termination but also that any node reachable during the tree construction will be included in the structure.

For least-cost, we show that under broadcast timing assumptions, the resultant tree will be least-cost except in certain identified cases.

Safety. We give an invariant predicate, Inv, for the states in which the protocol has constructed or is construction a tree with bidirectional edges from the root, and show that it is closed under all program actions. We partition the fault actions into E_0 that preserve the invariant and E_1 that do not, and show that the E_0 actions in fact preserve the invariant.

We give a fault-state predicate, F, for the states in which the structure has an unresponsive child, or will have one once the protocol terminates. We show that all actions except T1 preserve F and that, from an invariant state, the E_1 actions satisfy F.

When a message is sent along an interface, we assume that it is delivered without delay at every node that it is connected to via that interface; but the process of actually receiving the message is a separate step. Hence the result of `broadcast` ('wave',sn) on pi is that the broadcast message is delivered to all nodes connected via pi. The predicate dwave.j.i.(k,msn) is true if node j has received a wave message on interface i from node k with session number msn. `rcv` ('wave',msn) from k on i receives the delivered message and falsifies dwave.j.i.(k,msn). For simplicity, interfaces are ignored in this proof, so the predicate just mentioned would is given as dwave.j.(k,msn). The proof would be essentially the same if interfaces were included.

We use the following functions and notation in the invariants and their proofs.

parpath(y,z) means there is a path based on the par relation from y to z. Note that because a node only has a single parent, the path is acyclic.

childpath(y,z) means there is a path based on the child relation from y to z. If parpath(y,z) holds then by conditions (1a) and (1b) of the invariant, there is also an acyclic path from y to z based on the child relation.

hs.x.y is equivalent to dnotify.x.y $\vee$ dack.y.x.

Invariant and Fault Predicates

```
Invariant Inv:

sn.B>0 ∧ ∀y : sn.y=sn.B : (
                Tree structure
(1a) (∀x : x∈child.y : (upn.x ∧ upl.x.y ∧ ∨ tsn.x=sn.B) ∧
(1b) (y=B ∨ y∈child.(par.y)) ∧ up.y
(1c) (parpath(y,B) ∧
                Handshaking
(2)  (∀x : sn.x<tsn.x=sn.B : ¬upl.x.y ∨
       ((dnotify.y.(x,msn) ∨ dack.x.(y,msn)) ∧ msn=sn.B)) ∧
                Actions
(3a) (∀x : dwave.x.(y,msn) : (msn=sn.B ∧ x∉child.y)) ∧
(3b) (∀x : (y,msn)∈q.x : (msn=sn.B ∧ x∉child.y)) ∧
(3c) (∀x : dnotify.y.(x,msn) ∧ msn=sn.B :
       (x∉child.y ∧ upn.x ∧ upl.x.y ∧ tk.x=y ∧ sn.x<tsn.x=sn.B) ∧
(3d) (∀x : dack.x.(y,msn) : x∈child.y ∧ upn.x ∧ upl.x.y) ∧ tk.x=y)
         ) ∧
                Sequence number
(4a) (∀x :: sn.x≤tsn.x≤sn.B) ∧
(4b) (∀x,y : (y,msn)∈q.x : msn≤sn.B)

Fault states FS:

sn.B>0 ∧ ∃x,y :: sn.y=sn.B ∧
(F1) (parpath(y,B) ∧ childpath(y,B)) ∧
(F2) x∈child.y ∧
       ¬(upn.x ∧ upl.x.y ∧ (sn.x=sn.B ∨ tsn.x=sn.B) ∧ par.x=y)
```

We say that a *c*hild acknowledges its parent if it is contained in the parent's childset, if it points to the parent in its par variable, and if it and the link to the parent are up.

The key predicates are (1a) and (1c). (1a) indicates that children of tree nodes must either acknowledge the parent, or will do so when a pending ack message is processed. (1b) indicates that a tree node's parent acknowledges it. (1c) indicates that a proposed parent will satisfy (1a) when a pending notify message is processed.

Preservation of Invariant

For each action, we assume as usual that Inv holds in the pre-state and show that it continues to hold in the post-state.

Action T1. In the post-state, sn.B is larger than any other sequence number in the system. Hence it suffices to show that the invariant holds for y=B.

(1a). This is satisfied since in the post-state since child.B=$\emptyset$.
(1b). This is satisfied since y=B.
(1c). This is satisfied since y=B.
(2). The condition for (2) is false in the post-state since sn.y=sn.B and sn.B is larger than tsn.x for all other nodes. Hence (2) is satisfied.
(3a). If (3a) holds in the pre-state for some x then it holds in the post-state. If not, suppose dwave.x.(y,msn) holds in the post-state. It could only have been the result of this action. Since the action sends sn.B and since child.B=$\emptyset$ in the post-state, (3a) is satisfied.
(3b). The action does not affect the variables of this condition.
(3c). The action does not affect the variables of this condition.
(3d). The action does not affect the variables of this condition.
(4a). Since (4a) holds in the pre-state and since the action increases the session number, (4a) holds in the post-state.
(4b). The action does not affect the variables of this condition.

Action T2. This action receives a wave message and either enqueues it or ignores it.

(1a). The action does not affect the variables of this condition.
(1b). The action does not affect the variables of this condition.
(1c). The action does not affect the variables of this condition.
(2). The action does not affect the variables of this condition.
(3a). The action does not affect the variables of this condition.
(3b). Since the action only modifies the variables of id, it suffices to consider x=id and y=j. Suppose sn.j=sn.B. In the pre-state, by (3a), msn=sn.B and id$\notin$child.j. By (4), msn is as large as any session number in the system. Hence the program action will place (msn,j) in the queue (after resetting the queue if necessary to eliminate lower session numbers), with msn=sn.B. Since the child set of j is not modified, (3b) is satisfied.
(3c). The action does not affect the variables of this condition.
(3d). The action does not affect the variables of this condition.
(4a). The action does not affect the variables of this condition.
(4b). In the pre-state, msn$\leq$sn.B by (4a). Hence in the post-state, if (y,msn)$\in$q.id, (4b) is satisfied.

Action T3.
It suffices to consider x=id and y=k.

(1a). The action does not affect the variables of this condition.
(1b). The action does not affect the variables of this condition.
(1c). The action does not affect the variables of this condition.
(2). In the post-state, sn.id¡tsn.id=sn.B. By (1b), y is up. If the link is down then (2) is satisfied; otherwise, the action will send the message and satisfy (2).

(3a). The action does not affect the variables of this condition.
(3b). The action does not affect the variables of this condition.
(3c). If y and the link to y is up, the condition of (3c) will be satisfied in the post-state. In the pre-state, x is not a child of y by (3b), and this is preserved by the action. Since the action sets tk.x to y, (3c) is satisfied.
(3d). The action does not affect the variables of this condition.
(4a). Since (4a) holds in the pre-state, it holds in the post-state.
(4b). The action does not affect the variables of this condition.

Action T4. This action receives a notify message and, if the message's session number is the same as the node's, responds with an acknowledgement, adding the sender to its child set.

It suffices to consider x=id and y=k.

(1a). In the post-condition, x is a child of y. By (3c), x and the link between x and y are up, and tsn.x=sn.B in the pre-state. This is preserved, so (1a) is satisfied.
(1b). By (3c), sn.x¡sn.B, so the condition of (1b) is false and (1b) is trivially satisfied.
(1c). The action does not affect the variables of this condition.
(2). By (3c), x and the link between x and y are up. Since (2) is true in the pre-state, it holds in the post-state, with dack replacing dnotify.
(3a). The action does not affect the variables of this condition.
(3b). The action does not affect the variables of this condition.
(3c). The action does not affect the variables of this condition.
(3d). By (3c), tk.x=y in the pre-state, and x and the link between x and y are up. These are preserved and hence (3d) holds.
(4a). The action does not affect the variables of this condition.
(4b). The action does not affect the variables of this condition.

Action T5. This action receives an acknowledgement from k, and, if the message's sequence number is the same as that of the last wave, makes k its parent, and propagates the wave.

It suffices to consider x=id and y=k except for (2).

(1a). The action does not affect the variables of this condition.
(1b). By (3d), in the pre-state, x is a child of y. In the post-state, the parent of x is y and sn.x=sn.B. Hence (1b) holds.
(1c). Since (1c) is true in the pre-state, it is true in the post-state.
(2). In the post-state, sn.x=tsn.x, so the condition is false and (2) holds trivially.
(3a). Since child.id is emptyset in the post-state, and since sn.x=sn.B, (3a) holds.
(3b). The action does not affect the variables of this condition.
(3c). The action does not affect the variables of this condition.
(3d). The action does not affect the variables of this condition.

(4a). Since (4a) holds in the pre-state, it holds in the post-state.
(4b). The action does not affect the variables of this condition.

Action T6. A timeout occurs when a node is handshaking over a down link or with a down node. This does not affect the tree structure if the other node is not in the current session or if the handshake failed to begin since the link was down.

It suffices to consider x=id and y=tk.id.

(1a). If x is a child of y in the pre-state, then by (1a), the guard of the action is disabled.
(1b). The action does not affect the variables of this condition.
(1c). The action does not affect the variables of this condition.
(2). The action at most falsifies the condition of (2), so it is satisfied.
(3a). The action does not affect the variables of this condition.
(3b). The action does not affect the variables of this condition.
(3c). If the condition of (3c) holds in the pre-state then the guard of the action is disabled.
(3d). If the condition of (3d) holds in the pre-state then the guard of the action is disabled.
(4a). This is preserved.
(4b). The action does not affect the variables of this condition.

Action E1 when sn.id≠sn.B ∧ ¬(∃y : sn.y=sn.B : (dnotify.y.(id.msn) ∨ dack.id.(y.msn)) This expresses that to avoid violating the invariant, it must be the case that a node fails only when it is not in the current session and it is not handshaking with another node that is in the current session.

(1a). Suppose sn.id¡sn.B and tsn.id=sn.B. Then by (2), a handshake is in process, so the action is not enabled and (1a) is preserved.
(1b). Since sn.id¡sn.B, the condition of (1b) is trivially satisfied.
(1c). Since parpath consists only of nodes in the current session, (1c) is trivially satisfied.
(2). Suppose the condition of (2) holds. If the link between x and y is down then x can be failed and still satisfy (2). Otherwise, there is a handshake in process and the action is not enabled.
(3a). The action does not affect the variables of this condition.
(3b). The action does not affect the variables of this condition.
(3c). If a handshake with a node in the current session is in process then the action is not enabled. Otherwise, the condition of (3c) does not hold and so is trivially preserved.
(3d). If a handshake with a node in the current session is in process then the action is not enabled. Otherwise, the condition of (3d) does not hold and so is trivially preserved.
(4a). The action does not affect the variables of this condition.

(4b). The action does not affect the variables of this condition.

Action E2 when ¬((sn.id=sn.B ∧ sn.k=sn.B ∧ k=par.id) ∨ ((dnotify.id.(k,msn) ∨ dack.k.(id,msn)) ∧ tk.id=k ∧ sn.k=sn.B)) This expresses that to avoid violating the invariant, the linked nodes in question must not be in the current session; or, the node must not be handshaking with a node in the current session.

(1a). If the condition for (1a) holds and id is not in the current session, then it must be the case that sn.id¡tsn.id=sn.B and by (2), a handshake is in process, so the command is not enabled.
(1b). If id is in the current session then the command is not enabled and (1b) is preserved.
(1c). Since parpath consists only of nodes in the current session, (1c) is trivially satisfied.
(2). Suppose the condition of (2) holds. If the link between x and y is up there is a handshake in process and the action is not enabled.
(3a). The action does not affect the variables of this condition.
(3b). The action does not affect the variables of this condition.
(3c). If the condition holds then a handshake is in process and the action is disabled.
(3d). If the condition holds then a handshake is in process and the action is disabled.
(4a). The action does not affect the variables of this condition.
(4b). The action does not affect the variables of this condition.

Action E3. This restarts a failed node. Suppose the invariant holds and a node has failed. This means that the node is not part of the tree structure. When the node restarts, it session number is zero, which means it is not part of the tree structure. Hence the invariant is preserved for node restarts.

Action R1. This action is for liveness, not safety.

Violation of the Invariant.

For each action, we assume that Inv holds in the pre-state and show that FS holds in the post-state.

Action E1 when sn.id=sn.B ∨ (∃y : sn.y=sn.B : hs.id.y.) This expresses that to violate the invariant, it must be the case that a node fails only when it is in the current session or it is handshaking with another node that is in the current session.

We assume that B does not fail, so id≠B. By the invariant, id has a parent. Let y=par.id and x=id. By the invariant, parpath(y,B) and childpath(y,B) hold in the pre-state, so (F1) holds in the post-state. (F2) holds in the post-state because (1a) is violated.

Action E2 when (sn.id=sn.B $\wedge$ sn.k=sn.B $\wedge$ k=par.id) $\vee$ (hs.id.k $\wedge$ tk.id=k $\wedge$ sn.k=sn.B) This expresses that to violate the invariant, a link failure occurs with a parent when both are in the current session, or during handshake with a node in the current session.

Let y=k and x=id. By the invariant, parpath(y,B) and childpath(y,B) hold in the pre-state, so (F1) holds in the post-state. (F2) holds in the post-state because (1a) is violated.

Action E3. As noted earlier, a restart on a failed node cannot violate the invariant.

Action R1. As noted earlier, this action is for liveness, not safety.

Preservation of Fault States.

Suppose in some state that there is from the root a path (consisting of nodes from the current session) that leads to a node with a false child. The only actions that might affect this are node or interface failures that occur on the path.

Suppose a node failure occurs on the path. By assumption, the node cannot be the root. Hence the failure will the parent of the failed node to have a false parent, so the resulting state still satisfies the fault state predicate. The same reasoning holds for an interface failure.

Proof of Liveness. We show two things in this proof.

1. Termination. Eventually all guards are false.
2. Spanning. Suppose node X is neighbor to a node Y that is in the current session and that is propagating the wave. If X remains a neighbor to Y, then X will join the tree.

By "neighbor" we mean that X and Y share a link and that both are up. Note that the condition does not guarantee that Y will be X's parent; merely that X join the tree with some parent.

To give a variant function, we begin by forming the set of all possible acyclic paths from B to every other node, based on the link relation. It does not matter whether the links or nodes involved are up or down. We give a variant function for each path; the overall variant is the sum of the path variant functions.

The idea behind the path variant function is that when B begins a new session, the wave is a certain distance from a given node along each path that terminates with that node. Each action of the program either (a) moves the wave closer to some node along some path, or (b) the node joins the tree, or (c) the wave dies out because of failure. In the second case, the variant functions for all paths that terminate in the node assume a value of zero. In the third case, the path variant function assumes a value of zero.

We show that the path variant function eventually reach zero for each path, so the variant function eventually reaches zero, at which time the protocol has terminated and, if a tree was formed, it is spanning in the dynamic sense mentioned above.

To calculate the distance a wave is from a node along a path, we have to take into account the fact that (a) the wave will be enqueued before it is propagated and (b) a handshake must occur. Since a node can be neighbor to N-1 other nodes, where N is the number of nodes in the system, it can take N-1+4=N+3 actions before the node is propagated (N-1 for the queue, 1 to receive a delivered wave message into the queue, 1 to send the notify, 1 to receive the notify and send the acknowledgement and 1 to receive the acknowledgement and propagate the wave).

Let X be a node and let P(X) be a path from X to the root. We calculate the path variant function PV(P(X)) as follows.

- If there is no wave message for the current session on P(X) then then PV(P(X))=0.
- If X is in the current session then PV(P(X))=0.
- If there is a wave message for the current at node Y on P(X) then let d be the number of links from X to Y. Consider the following subcases.
  - If the wave message has been delivered to Y but not yet been received then PV(P(X))=d(N+3)+(N+3).
  - If the wave message is at position e in the queue (where e is between 0 and N-1) then PV(P(X))=d(N+3)+(e+3).
  - If Y has sent the notification to Z but it has not been received then PV(P(X))=d(N+3)+2.
  - If Z has received the notification and sent the acknowledgement to Y but it has not yet been received then PV(P(X))=d(N+3)+1.
  - If Y has received the acknowledgement and propagated the wave then PV(P(X))=d(N+3).

At each point, a link or node fault can cause the communication to fail and the wave message dies out on all paths that use that link or node.

Let V be the sum of PV(P(X)) for all P(X). It should be clear that each action of the tree construction protocol except T1 reduces PV(P(X)) for some node X. Since paths are finite, eventually each node X either joins the current session or the wave message dies out. Hence eventually all PV(P(X)) will be zero.

Thus the protocol terminates and satisfies the dynamic spanning condition.

Proof of Least Cost. Let ci be the cost of interface i, and let di be the delivery+handshake time. We assume that for interfaces i and j, if di<dj then ci<cj.

A tree that is formed might not be least-cost if a wave message arrives at node X via interface i from Y but the interface fails before handshaking begins. In this case, X will attempt a handshake with Y via i and it's failure will eventually be detected by a timeout, in which case the next wave message that arrives will be attempted. This delay can result in the tree not being least-cost.

From the structure of the protocol, it can be shown by a simple induction that if handshaking succeeds for the first wave message that arrives at a node then the tree constructed will have least-cost paths.

Initially the tree consists of only the base station, which has no edges and hence is trivially least-cost. Because di<dj implies ci<cj, the first delivered message will be least-cost and the node that joins will be over a least-cost edge. Hence the least-cost path property is preserved.

B Kansei Testbed

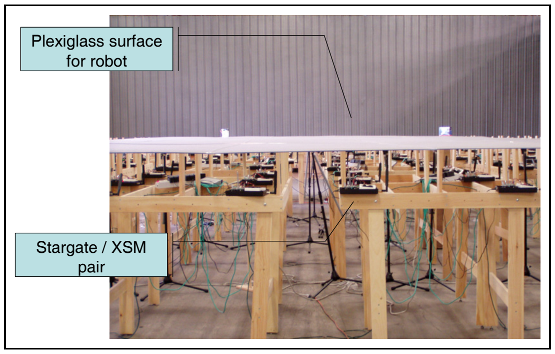


Fig. 5. The Kansei Testbed

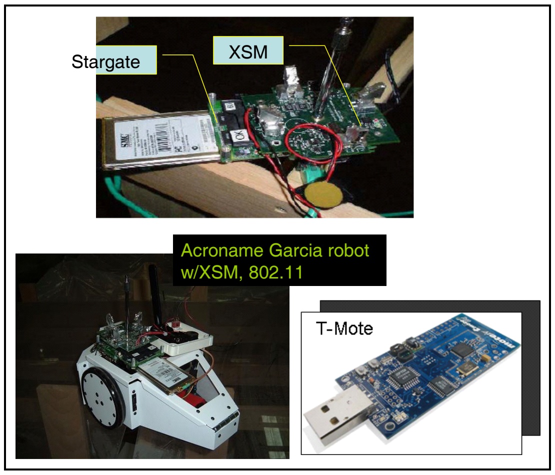
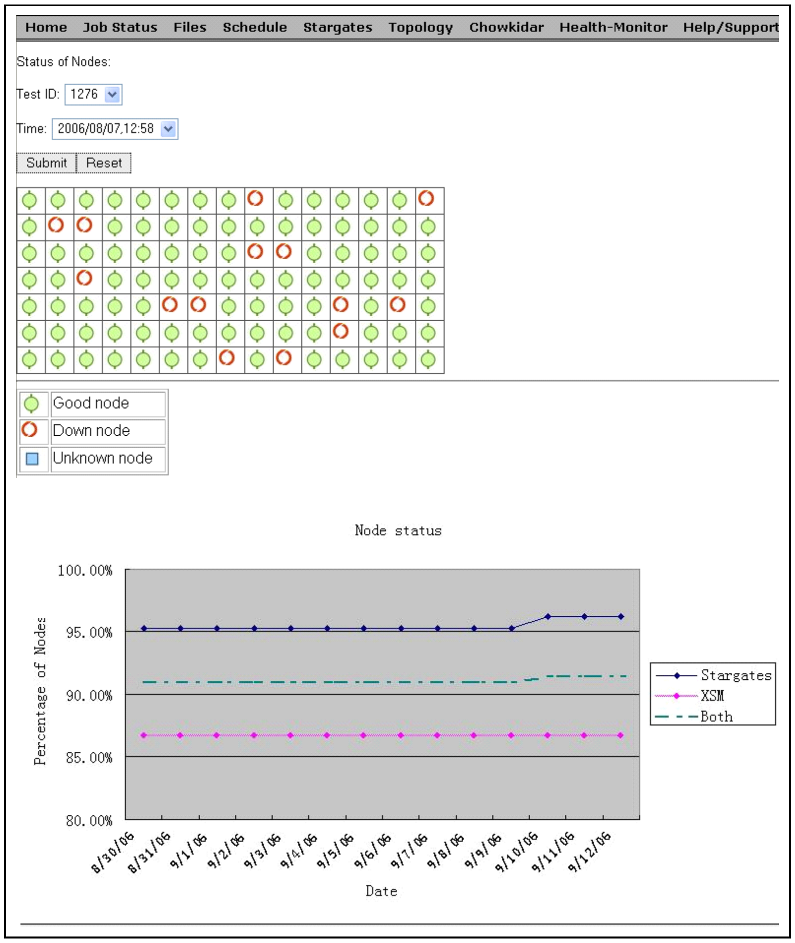


Fig. 6. Kansei Testbed Nodes

C Chowkidar Screenshots



Status of Nodes:

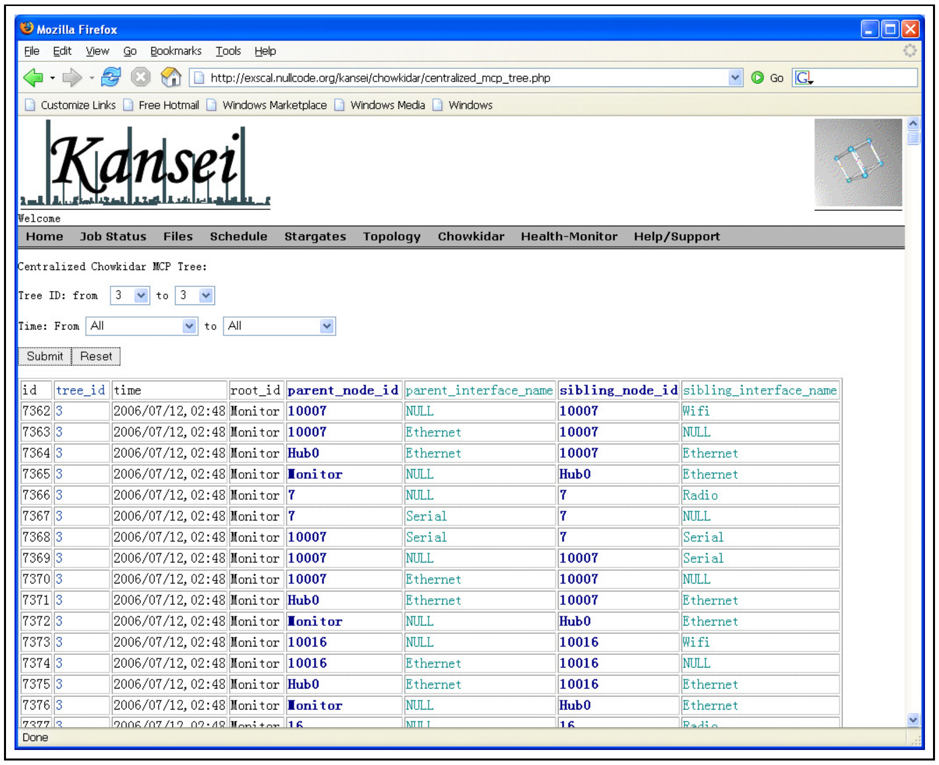
Test ID: 1276

Time: 2006/08/07,12:58

🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	⭕	🟢	🟢	🟢	🟢	🟢	⭕
🟢	⭕	⭕	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢
🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	⭕	⭕	🟢	🟢	🟢	🟢	🟢
🟢	🟢	⭕	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢
🟢	🟢	🟢	🟢	🟢	⭕	⭕	🟢	🟢	🟢	🟢	⭕	🟢	⭕	🟢
🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	🟢	⭕	🟢	🟢	🟢
🟢	🟢	🟢	🟢	🟢	🟢	🟢	⭕	🟢	⭕	🟢	🟢	🟢	🟢	🟢

🟢	Good node
⭕	Down node
🟦	Unknown node

Fig. 7. Chowkidar: Node Status



Centralized Chowkidar MCP Tree:

Tree ID: from 3 to 3

Time: From All to All

id	tree_id	time	root_id	parent_node_id	parent_interface_name	sibling_node_id	sibling_interface_name
7362	3	2006/07/12, 02:48	Monitor	10007	NULL	10007	Wifi
7363	3	2006/07/12, 02:48	Monitor	10007	Ethernet	10007	NULL
7364	3	2006/07/12, 02:48	Monitor	Hub0	Ethernet	10007	Ethernet
7365	3	2006/07/12, 02:48	Monitor	Monitor	NULL	Hub0	Ethernet
7366	3	2006/07/12, 02:48	Monitor	7	NULL	7	Radio
7367	3	2006/07/12, 02:48	Monitor	7	Serial	7	NULL
7368	3	2006/07/12, 02:48	Monitor	10007	Serial	7	Serial
7369	3	2006/07/12, 02:48	Monitor	10007	NULL	10007	Serial
7370	3	2006/07/12, 02:48	Monitor	10007	Ethernet	10007	NULL
7371	3	2006/07/12, 02:48	Monitor	Hub0	Ethernet	10007	Ethernet
7372	3	2006/07/12, 02:48	Monitor	Monitor	NULL	Hub0	Ethernet
7373	3	2006/07/12, 02:48	Monitor	10016	NULL	10016	Wifi
7374	3	2006/07/12, 02:48	Monitor	10016	Ethernet	10016	NULL
7375	3	2006/07/12, 02:48	Monitor	Hub0	Ethernet	10016	Ethernet
7376	3	2006/07/12, 02:48	Monitor	Monitor	NULL	Hub0	Ethernet
7377	3	2006/07/12, 02:48	Monitor	16	NULL	16	Radio

Fig. 8. Chowkidar: Least Cost Path Tree