Chowkidar: A Health Monitor for Wireless Sensor Network Testbeds

Sandip Bapat, William Leal, Taewoo Kwon, Pihui Wei, Anish Arora
Department of Computer Science and Engineering
The Ohio State University, Columbus, Ohio, USA
Email: {bapat, leal, kwonta, weip, anish}@cse.ohio-state.edu

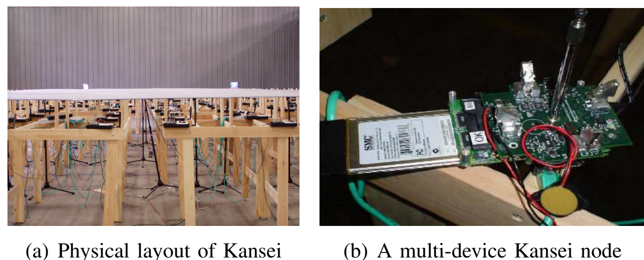
***Abstract*— Wireless sensor network (WSN) testbeds are useful because they provide a way to test applications in an environment that makes it easy to deploy experiments, configure them statically or dynamically, and gather performance information. Sensor data collected in the field can be replayed on nodes, and new ways to process the data can be tested easily. Testbeds are rapidly growing in size, with hundreds or thousands of devices, and testbed services are also becoming richer and more complex. Due to their size and complexity, faults can (and do) occur in these testbeds, affecting the outcomes of experiments. Awareness of testbed health status is important to both testbed administrators charged with maintaining functional services, and users who prefer to use healthy devices and like to know if there are any failures during their experiments.**

Based on our experience with Kansei, a large WSN testbed at Ohio State, we identify use cases that motivate the design of Chowkidar, a health monitoring facility. Key among these are: monitoring as a service that operates independently of users to provide up-to-date testbed status information; monitoring of heterogeneous testbed devices and networks; distinguishing between node and interface failures; and diagnosing common-mode failures such as power supply or Ethernet hub failure. We present in this paper, a centralized and a distributed Chowkidar protocol that reliably monitor the health of large, heterogenous WSN testbeds. We present experimentally measured Chowkidar performance as well as real experiences and lessons learnt from the integration of Chowkidar with Kansei, including feedback from both testbed users and administrators who have found Chowkidar to be a useful tool for improving the accuracy and efficiency of testbed experimentation and maintenance.

I. Introduction

Wireless sensor networks (WSNs) have gained in popularity, due to their potential for use in a variety of applications such as perimeter security and intrusion detection, structural monitoring, industrial sensing and control, medical applications, etc.; however developing and fielding one is hard. Simulations are a useful way to debug code and get basic protocols working, but they do not take into account the realities of radio communication, power consumption, unanticipated faults, and the like. Hence WSNs tested via simulations often do not work when deployed in the field. On the other hand, experimenting with a field-deployed WSN is not practical due to significant time and labor overheads. When something does not work in a WSN application, understanding why can be difficult since memory, power, bandwidth and reliability limit a developer's ability to instrument the network. A testbed, designed to support experimentation with actual devices in a realistic environment, provides an effective compromise between simulation and deployment that can speed WSN development by providing a supporting infrastructure to run, configure and monitor experiments.

[1]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.



(a) Physical layout of Kansei (b) A multi-device Kansei node

Fig. 1. The Kansei testbed at Ohio State.

Key to testbed efficacy is a reliable infrastructure through which deployment, configuration, monitoring and data retrieval for an experiment can be done with minimal interference with the experiment itself. Large testbeds have hundreds or thousands of devices, using different types of hardware and software. One example of such a large, multi-platform testbed is Kansei [1], which we have developed at Ohio State and is shown in Figure 1. Kansei currently houses several hundred WSN devices of different types, whose characteristics are listed in Table I.

Device type	XSM	TelosB	Stargate
Processor	4MHz	8MHz	400MHz
RAM	4KB	10KB	32MB
OS	TinyOS	TinyOS	Linux
Interfaces	CC1000, Serial	CC2420, USB	Ethernet, 802.11b, Serial, USB
Bandwidth	38.4kbps	250kbps	11Mbps

TABLE I
Different platforms in Kansei.

A. The Need for Health Monitoring

The use of WSN testbeds is rapidly evolving from simple testing and debugging of protocols to complex, multi-phase experimentation in which experiments are scripted so that when a particular run completes, the data generated is analyzed and a new set of parameters, likely to give better results, is automatically selected and the experiment is re-run. Such scenarios are common in testing routing or MAC protocols where many different parameters such as queue length, backoff intervals, and the like may have to be tuned to select the combination that works best. At present, Kansei already supports interactive experimentation in which a user can visualize, in real time, the data produced by an experiment and select new parameters that can then be injected by Kansei into the experiment. Automated execution of scripted experiments will soon become part of Kansei's scheduling and experiment management service.

Given the relatively unreliable nature of WSN devices, faults may occur before or during an experiment that affect the quality of data produced in a run. Even if devices are reliable, software bugs or incorrect device configurations could lead to faults. While designing, maintaining and experimenting using Kansei over the last two years, we have encountered a variety of faults, which we believe are also applicable to other WSN testbeds. These faults can be categorized as follows.

- *Device fail-stop faults.* A fail-stop results in the complete failure of a device. Fail-stops may occur as a result of hardware failure or due to software crashes that render the device completely unresponsive. The impact of a fail-stop fault may depend on the type of the affected device in heterogenous testbeds such as Kansei. For example, the fail-stop of an XSM mote simply results in that mote being unavailable for user experimentation. However, the fail-stop of a Stargate has much more impact since a Stargate is used by Kansei to program and log data from XSM and TelosB motes that are attached to it via their serial and USB interfaces. Similarly, the fail-stop of an Ethernet hub results in loss of wired connectivity to all of its attached Stargates and in turn their attached motes. Since the wired network is used by Kansei for instrumentation and data retrieval during an experiment, these fail-stops render the affected testbed regions unavailable.

- *Network interface faults.* A network interface fault at a device results in the device being unable to communicate over that interface. Network interface faults may occur due to driver failure or loose hardware connections such as an unplugged Ethernet cable, an unseated 802.11b wireless card or a detached Stargate-mote connector. Since most testbeds have separate experimentation and instrumentation interfaces, the failure of a single network interface does not render a device unreachable. Failure of all its network interfaces, however, results in a device being partitioned from the testbed.

- *Software faults.* A software fault results in a physically correct device or network interface being driven into a corrupted or bad state. These faults may occur due to bad or buggy code or due to misconfigured software. For example, a user program may change the radio power level to a very low value or change the transmission frequency, so that neighbors cannot receive radio messages sent by this device. Another type of configuration fault occurs if the Stargate or its attached mote changes the configuration of some pin in their serial connector to an incompatible state.

If these faults are not detected and accounted for while analyzing experimental results, users or automated test programs that consume them could end up deriving incorrect conclusions or making incorrect parameter choices that adversely affect the performance of subsequent runs. It is therefore imperative for health status information to be available before, during and after an experiment so that the results produced by a testbed experiment can be accurately analyzed.

In addition to node health status, diagnostic services provided by monitoring can help administrators distinguish between a testbed fault and an error caused by the experiment itself. This is important since some faults may have the same visible effect as others. For example, a 802.11b wireless driver process crash has the same effect as the card becoming unseated. Diagnosis of such faults can help administrators determine the appropriate correction actions needed to restore the testbed to a fully functional state. Knowing, for example, via alternate network interfaces that the wireless driver has crashed, an administrator could simply reload it, whereas an unplugged card would require the administrator to go to the testbed and physically re-insert it.

We therefore developed Chowkidar, Hindi for "watchman", as a service that provides health data about testbed resources, periodically as well as on demand. Users (or a testbed scheduling service) can use the information provided by Chowkidar to ensure that experiments run only on working devices by checking health before and after an experiment. They can also better assess experimental results knowing whether or not some failures occurred during the experiment. Administrators can use Chowkidar to learn about testbed faults and any available diagnostic information to help determine the most effective response.

B. Monitoring Requirements

The general network monitoring problem may be stated as follows: to identify which network objects are functioning correctly, and, for those that are not, to identify them and so far as possible, indicate why they are not functioning. Objects typically include network resources such as links, nodes, and so forth, but may also include things such as application components and processes.

In our practical experience with Kansei testbed, various

recurring use cases have caused us to add a number of specific requirements to monitoring WSN testbeds.

- *Reliability.* For monitoring results to be useful, they should be reliable. Reliability has two important dimensions; first, the reported results should be consistent with the actual state of the testbed and second, they should be complete so that if a node is working and is reachable, this status should be known to the monitor. Due to resource constraints such as limited bandwidth and energy, WSNs often use sampling-based monitoring where the overall state of the network is estimated based on data received from a subset of nodes. The goal of a testbed health monitoring service however, is to provide ground truth information against which users can compare obtained experimental results. We therefore require that the status of each resource in the network be monitored and that these results be reliably exfiltrated.

  Reliability also implies that monitoring must be independent of an experiment's semantics or communication structure; otherwise design or implementation errors in an experiment could yield incorrect monitoring results.
- *Efficiency.* Regardless of whether a monitor collects health status from some or all nodes, monitoring should be both time and energy efficient. By this we mean that monitoring must complete quickly, using few messages.
- *Heterogeneity.* As exemplified in Table I, WSN testbeds may have different hardware and software platforms with varying capabilities and limitations. A monitor must therefore handle heterogeneous devices and networks. This might include PCs, embedded Linux systems such as Stargates and motes such as Mica2s, XSMs or TelosBs, as well as a variety of networks, including mote radio, WiFi, Ethernet, and others. There can be several instances of each network (such as multiple Ethernets). A monitor must be able to check the status of each of these.
- *Diagnosability.* Administrators need to be able to distinguish between complete device failure and interface failure. In the former case, devices usually have to be physically repaired or replaced; in the second, a remote correction might be possible. Hence the monitor must distinguish between these failures.
- *Adaptability.* WSN testbeds keep evolving due to factors such as device failures and replacements, new technology and changing user requirements. Testbed configurations may also change as a result of rewiring and physical relocation. A monitor must therefore not be dependent on any particular testbed architecture and must be able to easily accommodate different types of networks and devices and adapt to changes in network configuration with minimal efforts.
- *Usability.* Monitoring results must be available centrally to users and administrators. Administrators need status information periodically to detect permanent failures or identify failure patterns while users want to know the network status before and after an experiment. Hence monitoring must be performed both automatically or on demand, targeting specific devices as appropriate.
- *Composability.* Users may be interested in monitoring node health during an experiment. Hence monitoring should be easy to compose with user applications. At the same time, a monitor must have at most bounded interference with experiments during concurrent operation. This is of particular concern for radio networks, but can affect others such as Ethernet as well.

Existing monitoring approaches such as Motelab [2], SNMP [3], SNMS [4], fault tracing [5] and Sympathy [6] satisfy some of these requirements but none of them satisfy all of them. In particular, none distinguish between node and interface faults, and none are heterogeneous. This is discussed in more detail in Section IV. It was because of these limitations that we developed Chowkidar.

Although these monitoring requirements are desirable, not all testbeds might support them. We therefore identify the following set of assumptions about testbeds in general for Chowkidar to realize the monitoring requirements listed above.

- There is an organizational structure that permits automatic, unattended monitoring. If we want monitoring to be conducted when nodes become free, for instance, there must be a way for the monitor to identify those nodes. In Kansei, the scheduling service maintains a database of node status that Chowkidar accesses.
- To avoid interference with experiments running concurrently on the testbed, radio-based communication (including mote radio and 802.11 WiFi) has a channel dedicated to monitoring; or else policies are structured in such a way as to disable certain kinds of monitoring when necessary to prevent interference.

C. *Organization of the Paper*

The rest of the paper is organized is follows. Section II presents details of Chowkidar including its fault model and centralized and distributed versions of its monitoring protocol. Section III describes implementation details of Chowkidar including performance results measured experimentally on the Kansei testbed, integration with Kansei and real user experiences. Section IV discusses related work. Finally we discuss future work and conclude in Section V.

II. Monitoring a WSN Testbed with Chowkidar

In this section, we first describe the fault model for Chowkidar and our approach for diagnosing failure dependencies. We then present two Chowkidar protocols: the first uses centralized control while the second one is distributed. Recall

from Section I-B that reliability and efficiency are the key requirements for WSN testbed monitoring, hence we especially focus on these aspects in our discussion.

Although the two protocols differ in a number of ways, they both have the following characteristics.

- Monitoring information is collected and evaluated centrally at a base station. The base station is aware of the topology of the network, and knows which nodes are in use by experiments and which are not. This information is used to perform monitoring, to assess the results, and to provide current and historical status.
- Monitors are correct in the presence of node and link fail-stops and restarts that occur during the run or between runs. A given collection either gives a consistent result (corresponding to testbed state that existed during the run) or reports failure. In the latter case, the monitor can be run again.
- All devices are explored for reachability along all available networks, with a typical path consisting of multiple networks.
- Paths to resources are least-cost, where the cost of a network link is assigned by the administrator, taking into account factors such as bandwidth, reliability, and interference.
- Monitoring runs are done periodically and, at the request of the user or the testbed scheduler, can be done on demand.
- Between monitoring runs, no resources are used except for components that listen for monitoring messages.

As a minimum, we want Chowkidar to monitor testbed resources that are not in use. We assume that the testbed has a scheduling service with the following characteristics.

- When an experiment begins on some set of resources, Chowkidar is informed that those resources are in use.
- When an experiment ends, Chowkidar components are automatically loaded on those components and Chowkidar is informed that the resources are available.

In our Kansei implementation, Director is the scheduling service; it maintains an SQL database of resource status. Chowkidar accesses this database to assess resource status for monitoring.

Since Chowkidar is configuration-driven, it is testbed-independent. For a given testbed, communication components for each network on a node must be provided along with a configuration of the testbed as a whole. Other than this, no changes need be made to the architecture of Chowkidar to use it on another testbed.

A. Fault Model

We assume fail-stop faults with clean restart. When a node fails, it does not communicate on any of its interfaces. When an interface fails, the node cannot communicate on it. When a node restarts, it does so cleanly, reinitializing variables as required by the monitoring component. We assume an interface does not have state, so a restart is always clean.

A fault can happen at any time, including during a monitoring run. If this happens, we want the monitoring either to report the failure so that it can be restarted, or we want the collection to be consistent in the sense that the state reported actually occurred in the system during the run.

For example, suppose a reachable node's status has been confirmed and, during the balance of the monitoring run, it is not accessed again. If this node fails before the end of the monitoring run, the report (that the node is reachable) will be inconsistent with the final state of the system (in which it is not); but it will be consistent with the state of the system before the node failed.

On the other hand, if a node was found to be reachable at one point during the run but found to be unreachable later, then the run would terminate with an error value. If we assume that faults stop, or at least stop long enough, then an erroneous run can be rescheduled, and eventually it will succeed.

Links in a WSN can be unidirectional or bidirectional. While we do not require all links to be bidirectional, we do assume that a bidirectional path exists from the root, or the Chowkidar server, to each node. Thus, our protocols may try to use some unidirectional links and fail, but eventually they will discover a bidirectional path if it exists.

Wireless links are subject to interference in the presence of multiple concurrent senders. Interference affects link reliability and may result in message losses due to collisions. Message losses during exfiltration of health data from the testbed may affect monitoring reliability.

B. Diagnosing Dependencies

Basic monitoring gives reachability information about nodes and interfaces. If a node is reachable then it is up; but if not, we cannot automatically conclude its status, since it might be still be functional, in the sense that it would be reachable if placed in a suitable environment. Although unreachable, a node might be functional if the power is off, if all of its interfaces are not functional, or if it can only be reached through other nodes that are unreachable.

We have to consider the case of devices where checking status depends on some other device. These include "dumb" nodes such as Ethernet hubs that cannot be addressed directly as well as device interfaces and power supplies.

To know whether a dumb node such as an Ethernet hub is working or not we have to try to reach an attached node. Hence if some attached node is reachable via the dumb node then the dumb node is reachable and therefore up. However, if no attached node is reachable via the dumb node then either the dumb node is not functioning, all of the attached devices are not functioning, all or the interfaces of the attached devices

are not functioning, or some combination. Of course, there can be additional dependencies, since the interface of an attached node won't function if the node itself is not functioning, which in turn might be due to a power supply failure.

For power supplies, the supply is up if some associated node is up. If all associated nodes are not reachable then either the power supply is not functioning or all the nodes are unreachable for some other reason.

For interfaces, an interface on a node is reachable if it can be used to reach some other node; in this case, the interfaces are up as are both nodes and the link between the nodes. Although unreachable, an interface might be functional if its node is unreachable or if no other nodes are reachable via the interface.

Identifying these alternatives requires knowledge of the testbed topology. Intuitively, some alternatives are more likely than others. If a hub and all of its attached nodes are unreachable via the hub, it is more likely that it is the hub that is not functional. These intuitions can be used to order the alternatives to guide a testbed administrator. As part of future work we plan to gather data that relate alternatives to the actual diagnosis so that probabilities can be assigned based on historical data.

C. Centralized Chowkidar

The centralized version of Chowkidar performs a collection on the network that indicates which nodes are reachable. It also gives information about the status of interfaces. The main idea is that the base station, using configuration information and knowledge of nodes that are in use, attempts to construct a path to each node whose status is unknown, avoiding links that are down. The process terminates either when all nodes are confirmed as up or when there are no more paths to check.

Since a given collection does not depend on previous ones, the protocol handles both fail-stop and restart directly. If failure happens while the protocol is running that affects the consistency of the collection, it will be detected and the collection aborted. Restarts can happen at any time and will be included the next time a path is created that includes the resource.

The configuration information can be provided in two ways. In the "high atomicity" view, the configuration consists of nodes with network links between them. In this case, a collection will give all reachable nodes but may not check all interfaces. Alternatively, the configuration can be given with "low atomicity" in which interfaces are explicitly included. This forces Chowkidar to explore paths that include each interface, thus checking each one.

The following process is repeated, where initially, the status of each node and link is unknown.

1) Using testbed configuration information, construct a least cost path (LCP) tree that covers the free nodes of unknown status, using links that are not down, with the constraint that all leaves are nodes whose status is unknown. The tree can be empty, which means that all reachable nodes have been found, and we are done.
2) For each path, construct a probe message that contains the path. Send the message to the first node along the designated interface. As each node receives the message, it forwards it to the next node if any, and waits for a reply. If a node is the leaf in the path, it sends a reply back along the path. If a node that is waiting for a reply receives it, it forwards it on; if it does not arrive after a timeout (which can be calculated from the path), it replies on its own.
3) When a reply arrives at the base station, the knowledge of the path and the node that replied lets the base station identify the nodes and links that were reachable along that path; these are marked as "up". If some node other than the leaf replied then the downward link is marked as down and is effectively removed from the topology for the duration of the run.

   A network link involves a sender and a receiver; hence lack of responsiveness on the link implies that the sender's interface is down or that the receiver's interface is down or that the downward node node is down (or any combination). In any case, we consider the link as failed and do not use it again: it is removed from the configuration for the duration of the run.

If, during the run, a particular node or link was found to be up as the result of some probe but later when that node or link was reused, it was found to be down, then a node or link fault has occurred that affects consistency. In that case, we abort the run and restart. If the run did not abort with an error, then the collection is consistent. Hence, in the presence of restart or of fail-stop that does not affect consistency, centralized Chowkidar will terminate with a consistent collection. If a fail-stop happens that might affect consistency, it will terminate with an error.

As a protocol that checks reachability by exploring all paths sequentially, centralized Chowkidar does not scale well. Calculating an LCP tree takes $O(|N| \cdot |E|)$ time where N is the number of nodes and E is the number of edges per node. Since a network can be fully connected, the time complexity is $O(N^2)$. It results in $O(N)$ probes and, since the set of probes cover the nodes, there are $O(N)$ messages total. The number of times the LCP tree calculation process and probing has to be repeated depends on the pattern of failures. It can be as high as $O(N^2)$ in a network where half the nodes are down and each is directly linked to an up node; in this case, paths will be created from each up node to each down one. Hence time complexity can be as high as $O(N^4)$ and message complexity as high as $O(N^3)$.

D. Distributed Chowkidar

As described earlier, in the worst case pattern of failures, the time and message complexity of the centralized protocol can be quite high. From our experience with Kansei, a collection for centralized Chowkidar can take up to 10 minutes for the low atomicity case for 210 nodes. Kansei is in the process of growing, with up to 630 TelosB motes to be added in the near future, so clearly the centralized approach will not scale.

We have therefore developed a self-stabilizing distributed protocol [7] to be used as part of Chowkidar. This protocol solves an instance of 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. Our protocol differs from previous work in message-passing PIFs in two ways that are critical for the WSN model. First, it is message efficient in that it uses only a few messages per node, which is important given the resource constraints of WSNs. Second, it tolerates ongoing node as well as link faults, and their restart, which do indeed occur in WSNs, in contrast to requiring that faults stop during convergence.

Our distributed protocol first builds a spanning tree over the set of reachable nodes in the network. A key idea in this tree construction is a handshake between a node and its potential parent. At the start of an execution, the root (or the central Chowkidar server) broadcasts a wave message on all its outgoing interfaces with a session number higher than any used previously. When a node X receives a wave broadcast from another node Y with a higher session number, it asks Y to become its parent. Y records X as a child and sends an acknowledgement. Our protocol also forms LCPs from each node to the root by phasing the delivery of the wave messages: nodes forward a wave message on all outgoing links; however on links with lower cost, the messages are forwarded earlier than on links with higher cost. A node that is connected to the root through multiple paths will therefore receive a wave message on the LCP first as the total forwarding delay is proportional to the total path cost and select it.

If node or interface 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 subsequently run on the tree, each parent waits on its children to report before it reports to its parent; if the parent fails to hear from a child in a timely fashion, it initiates a failure message to the root, in which case a new tree is constructed. The acknowledgement process between the proposed parent and child combined with child timeouts 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. Under certain fault conditions, it may be possible for two nodes to consider the same node as their child; however this fault is detected during the PIF phase when one of them does not receive a report from that child. A formal protocol description can be found in [7] and proofs of its correctness in a related technical report [8].

When a tree formation or PIF phase 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. The message complexity is O(N); the time complexity depends on the height of the tree which is O(log N) in the normal case but can be O(N) in the worst case where link failures result in a single path through the network. If there are no failures once a correct tree has been constructed, subsequent PIFs will continue to use the same tree. Thus, for every such PIF, two messages per node are required: one to propagate the wave and one to return the feedback. Hence the message and time complexity are the same as for tree formation. 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 but before the acknowledgement is complete.

III. Chowkidar Implementation for Kansei

We have implemented both the centralized and distributed Chowkidar protocols for the Kansei testbed, though both implementations can easily be adapted to other testbeds with minor modifications. Our implementations span the different hardware and software platforms in Kansei listed in Table I. In this section, we first compare the performance of both protocols based on data collected from several experiments in Kansei. We then describe some important lessons learnt during the integration of centralized Chowkidar and some real experiences from both users and administrators of Kansei in using Chowkidar.

A. Experimental Results

To evaluate the performance benefits of using the distributed protocol over the centralized one, we ran a number of experiments using both implementations in our Kansei testbed. We ran both protocols on the same sets of nodes. In the initial experiments, we tested correctness in the absence of failures by simply executing the protocols on nodes that were known to be working. We then injected failures by killing Chowkidar processes on randomly selected nodes (the same nodes for both cases).

Table II shows the experimentally measured performance for a set of 25 nodes in Kansei. This data does not take into account the time taken to compute the paths in the centralized case as this is quite small on a powerful server. Recall that the distributed protocol first constructs a spanning tree over which subsequent PIFs can be collected, hence the total time for the distributed protocol is the sum of the times taken by each of these phases.

% of failed nodes	$\mathbf{T_{cent}}$	$\mathbf{T_{dist}}$	$\mathbf{T_{tree}}$	$\mathbf{T_{PIF}}$
0%	9s	7s	2s	5s
8%	54s	14s	8.5s	5.5s
20%	86s	16s	10.5s	5.5s
40%	153s	17s	11.5s	5.5s

TABLE II
PERFORMANCE COMPARISON ON A 25 NODE NETWORK.

As seen from the data, in the absence of faults, the performance of the centralized and the distributed protocols is quite comparable. However, the performance of the centralized protocol degrades substantially as the number of failures increases, even for a 25 node network. This is because the centralized protocol not only operates sequentially but also tries to explore all possible working paths in case of a failed node before giving up. By contrast, the distributed protocol finds existing paths concurrently instead of pruning failed ones sequentially, so its performance is only marginally affected. It should be understood though that the centralized protocol was inherently reliable due to its sequentially design that avoids interference losses whereas the distributed implementation had to be carefully tuned to select appropriate randomized backoff parameters to minimize network interference created by concurrent execution. The centralized protocol could also be parallelized and similarly tuned for reliability, but it is clear that it will not outperform the distributed one.

We also experimentally measured the scalability of both protocols by varying the network size, the results of which are shown in Table III.

# of nodes	% of node failures	$\mathbf{T_{cent}}$	$\mathbf{T_{dist}}$	$\mathbf{T_{tree}}$	$\mathbf{T_{PIF}}$
25	0%	9s	7s	2s	5s
50	0%	23s	9s	3s	5s
25	40%	153s	17s	11.5s	5.5s
50	40%	305s	23s	17s	6s

TABLE III
SCALABILITY OF CENTRALIZED VS. DISTRIBUTED PROTOCOLS.

The first two rows in the table indicate the completion times for both protocols in the absence of any injected faults while the last two rows indicate the completion time when (the same) 40% nodes, selected randomly in the network, are failed. The data clearly demonstrates that as the network size increases, the performance of the centralized protocol degrades much faster than the distributed one.

Another important point to note in the distributed case is that the PIF completion time increases only slightly as the failure rate and network size are increased. This is because the PIF completion time is a function of the depth of the constructed spanning tree. Also, since the same tree is used when there are no new failures, the PIF cost is amortized over multiple successive runs, hence if failures occur rarely, the average completion time for the distributed protocol is even smaller.

Our experiments thus show that when carefully tuned for reliability, the distributed protocol outperforms the centralized one and scales much better as both failure rate and network size are increased.

B. *Integration with Kansei*

As noted, Chowkidar is testbed-independent. As a case study, we have integrated its centralized implementation with the Kansei testbed, which satisfies the requirements mentioned earlier.

The Director service in Kansei is a distributed implementation that schedules and manages experiments, automatically terminating them when the reserved time has passed. When that happens, each Stargate activates Chowkidar's Stargate components and loads Chowkidar's mote components; since this happens locally, it does not depend on the base station and hence does not depend on reachability via Ethernet. At the base station, Director updates the status of nodes in a central database. When Chowkidar is scheduled to run, either periodically or upon demand, it accesses this database and checks free nodes. If Director needs a node for a scheduled experiment, it kills the Chowkidar components, rendering the node inaccessible to Chowkidar, and updates the database. If Chowkidar has completed probing the nodes just removed then there is no problem; otherwise, if the node was up, Chowkidar will note that it is no longer accessible and will terminate with error and restart.

There are various policy issues that require coordination between Director and Chowkidar. XSM and TelosB motes have several non-interfering radio channels available in their operational frequency band. Radio communication in Chowkidar can thus occur on a reserved frequency, avoiding interference with experiments. However, because Kansei is located in a warehouse with industrial neighbors, interference prevents Chowkidar from using a reserved frequency for 802.11b WiFi. To avoid interference, Director should note experiments that use the WiFi network so that Chowkidar can avoid using it. A similar policy issue concerns Ethernet in case Chowkidar's use of Ethernet might interfere with an experiment.

Since Stargates have substantially more resources than XSM and TelosB motes, it is reasonable to perform monitoring on them even when they are in use by an experiment. However, a particular experiment might prefer that Chowkidar not run during that time. Director and Chowkidar need to be set up so experimenters can indicate their preference.

Implementation of these policy issues is part of ongoing work.

C. *Experience with Chowkidar*

Since its integration with Kansei, users and administrators have been using Chowkidar quite actively for different reasons.

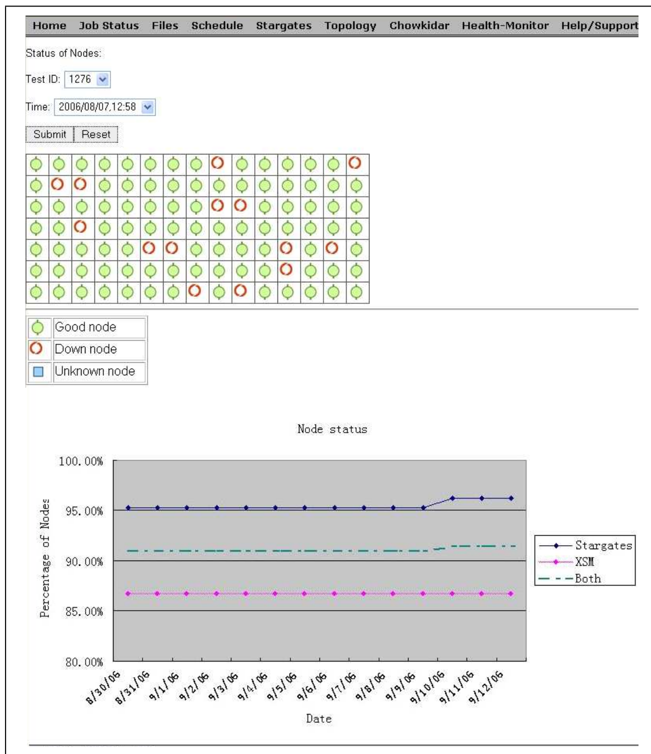


Fig. 2. VISUALIZATION OF KANSEI HEALTH USING CHOWKIDAR.

In this section, we describe our initial experiences and lessons learnt from user feedback after deploying Chowkidar.

Visualizing test results. At the end of a run, Chowkidar timestamps and stores the collected results in a database. The results are also displayed on a webpage [9] so that they are easily readable to users and administrators.

Figure 2 shows a screenshot of the output generated by Chowkidar which represents a high-level view of Kansei health. In this visualization, each Stargate and its attached motes are represented by a single logical node. Logical nodes with all devices working correctly are denoted by simple circles whereas if either of these devices or their interfaces have failed, the node is denoted by a bold, broken circle. Nodes whose status could not be monitored because no neighbors were reachable are denoted by squares. Users can learn more about exactly which devices and interfaces have failed by clicking on the corresponding circle or square. A graph shows recent history of the devices. As seen in the figure, the visualizer allows users to view the output of previous runs by specifying their id or timestamp. We are currently extending the visualizer to display the health history of a particular node. Since experiments can leave nodes in an inaccessible status, we plan to display cases where nodes failed immediately after an experiment.

Using Chowkidar results. In the past, Kansei users would schedule experiments on a set of nodes only to realize later that some of them were not working. This usually led to users having to retry several times before they finally ended up with a set of working nodes. However, users now check the latest Chowkidar output or invoke Chowkidar on-demand prior to scheduling experiments so their experiments always run on working nodes. By running Chowkidar on-demand after an experiment, users are also able to verify that no new failures occurred during their experiment, increasing confidence in the obtained data.

Chowkidar is also being used by Kansei administrators to diagnose failures. Previously, an administrator would execute a script to ping all Stargates and then individually diagnose the ones that did not respond. This approach is not only tedious but also does not work for non-IP based devices such as motes that do not respond to ping commands. However, using Chowkidar, Kansei administrators are now able to detect several new types of faults such as mote failure, failure of particular interfaces, etc. Even in cases where Chowkidar cannot definitively diagnose what fault has occurred, it provides enough information to the administrators to simplify manual debugging. Administrators are also using historical information in Chowkidar to identify failure-prone devices. For instance, if a particular mote oscillates between correct and failed states over many Chowkidar runs, it is highly likely that its serial connector to the Stargate may have become loose leading to disruptions in power supply to the mote.

Given that the implementation of Chowkidar is reliable, we have been able to use Chowkidar for diagnosing failures of the Kansei Director service, which keeps evolving as new features are added and so is occasionally subject to programming bugs. For instance, if an unusually high number of nodes are reported as failed by Chowkidar, it is likely that some Director component may have failed, causing the Chowkidar service itself to not be loaded correctly.

Monitoring predicates. After Chowkidar went online, we received feedback from several Kansei users and administrators about additional information they would like to be monitored. For example, testbed administrators were interested in monitoring whether the various Director processes were running correctly on Stargates while users wanted to know whether the SerialForwarder program used to send/receive TinyOS packets to/from motes was running throughout an experiment. As a result, we identified several new predicates besides node and network health that can now be monitored using Chowkidar.

IV. RELATED WORK

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; although a testbed itself is usually heterogeneous, existing support tools do not take this into account. Tools do not distinguish between the health of a node and the health of its interfaces. For some, the reliability is

too low to be useful and for others, there is a dependency on the communication structure of the application. Also, in most testbeds monitoring and the subsequent failure diagnosis requires explicit action on the part of a user or administrator and is not done automatically.

Traditional networks such as the Internet use standard protocols such as the Simple Network Management Protocol (SNMP) [3] 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. In WSN testbeds, there are often multiple paths to a node using alternative networks (such as mote radio), but SNMP's dependence on IP precludes its use.

Similarly, Motelab [2], Tutornet [10] and Orbit [11] 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 provide diagnostic information that can for instance help distinguish between node, interface and network device faults.

The Sensor Network Management System (SNMS) [4] provides networking support for WSNs via its own networking stack, including routing. SNMS allows network administrators to remotely query network devices and learn their status. However, experimental studies such as [12] have shown that reliability of SNMS does not suffice to provide accurate fault status, leading to false positives in detecting failures.

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. Sympathy monitors the flow of application traffic, evaluates the defined metrics, and communicates them to the base station using additional messages. This information is collected by an automated failure detector program at the base station, which tries to localize the type and the source of the faults in the network and notifies the user. A similar approach is used in [5] where the fault management system exploits not only the continuous data traffic flow in the network to piggy-back health information, but also uses the route update messages in the routing protocol to effect changes in routing paths for suspected nodes in order to trace failed nodes. These approaches, although similar to ours, are critically dependent on knowledge of application routing and traffic patterns. Further, in both approaches, monitoring is not conducted when an application is not running, which is a requirement for testbed health scenarios.

None of the protocols described above are designed to deal with heterogenous networks. Existing implementations of these protocols only work only for a homogenous network of certain types of devices. However, even if these implementations were adapted to span multiple platforms and networks, it would not be sufficient. This is because the different types of nodes and networks in a testbed each have different physical characteristics and resource constraints that can change with the nature of experiments running on them. For instance, the Ethernet network in the Kansei testbed is well-suited for reliably exfiltrating large amounts of data in a short time. Similarly, the XSM and Stargate nodes can be tuned to use different radio frequencies so that interference between health monitoring and ongoing experiments can be minimized. Dealing with heterogeneity thus requires knowing and adapting to the specific device characteristics and available resources, which is not addressed in existing protocols.

As described in Section II, the distributed Chowkidar protocol only requires O(N) messages per PIF wave. This is because nodes maintain information about child links in the spanning tree and thus by design aggregate their responses. Although it might be possible to extend Sympathy and SNMS to include aggregation, the protocols as defined require one message per node and therefore have a complexity of O(N log N) messages per collection.

V. Conclusions and Future Work

A reliable, heterogeneous and efficient health monitoring service is critical to successful maintenance and use of a wireless sensor network testbed. Chowkidar has proven to be a useful tool for assessing the testbed health. The centralized version is suitable for smaller testbeds while the distributed version works efficiently for larger ones.

Future work will mainly focus on three aspects; extending the functionality of Chowkidar, making Chowkidar more efficient and improving its usability. We identify the important tasks in each of these.

An important predicate that needs to be monitored, especially in WSNs, is the quality of radio links. Monitoring this predicate requires the exchange of several messages before an evaluation can be made. We plan to integrate a link estimation service so that Chowkidar can report on link quality in addition to basic "up"ness. The evaluated predicates can also provide feedback to Chowkidar itself, so that a node dynamically adjusts link costs depending on the estimation.

Our current Chowkidar implementation ignores the monitoring of sensors, which are an important resource in WSN testbeds. Monitoring sensor health is difficult due to several reasons. First, ground truth is often not available, even in a controlled testbed setting, so there is no absolute reference point for evaluating obtained sensor readings [13]. 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 variations is important. We plan to use robots that are part of the mobile Kansei platform, to help monitor the health of a sensor by generating a known signature in its neighborhood. Similarly, we wish to monitor actuator health using calibrated sensors.

At present, Chowkidar does not distinguish interface failure from misconfiguration of the interfaces (say, to disable the Ethernet driver). In general, however, misconfiguration can be detected by reading the status of device registers or environment variables, so we plan to add configurations to the list of predicates to be monitored by Chowkidar. When a device interface is misconfigured but the device is accessible via some other interface, this fact can be reported.

As network scale increases, the issue of bidirectional link reliability becomes more important. Towards improving the efficiency of Chowkidar in unreliable wireless environments, we will compare the performance of CSMA-based approaches combined with appropriate timing choices for backoffs, that provide probabilistic guarantees about accuracy with deterministically reliable schemes such as TDMA.

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. However, a broadcast might be heard by many neighbors and if they all report it, there will be excessive redundancy. Also, there may 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. Future research will focus on techniques such as data compression and in-network aggregation to improve the efficiency of collecting transmitter health.

Besides functionality and efficiency, we also plan to improve the usability of Chowkidar. Chowkidar presently monitors only those nodes that are not running a user experiment. Monitoring the health of nodes running an experiment is desirable from a user's perspective to improve confidence in the experiment outcome. We plan to address the concurrent execution of Chowkidar with a user experiment in two ways. First, we will provide a standard set of lightweight Chowkidar components, along with tools for easy integration of user and Chowkidar components. Second, we will define mechanisms whereby users can specify policies that dictate which and what fraction of available resources on a node running a user experiment can be used by Chowkidar for health monitoring. This will provide flexibility to users in controlling the interference between the experiment and health monitoring. Another interesting idea for future research is to investigate whether there is a systematic way to exploit the semantics of an application for monitoring while still offering correctness guarantees.

At present, the integration of Chowkidar with Kansei is one-way, since the information reported by Chowkidar is not used by Director. Future integration steps will involve Director using the output of Chowkidar to automatically select a set of nodes that are known to be good to run an experiment.

We also plan to design visualization and analysis tools that will help users and administrators better interpret monitoring results, including history, produced by Chowkidar. This includes distinguishing interface vs. device failures, the data required for which is being collected even in the existing Chowkidar implementations.

REFERENCES

[1] Anish Arora, Emre Ertin, Rajiv Ramnath, Mikhail Nesterenko, and William Leal. Kansei: A high-fidelity sensing testbed. *IEEE Internet Computing*, 10(2):35–47, March/April 2006.

[2] G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: A Wireless Sensor Network Testbed. In *4th Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, 2005.

[3] IETF. RFC 1157. www.ietf.org/rfc/rfc1157.txt.

[4] G. Tolle and D. Culler. Design of an Application-Cooperative Management System for Wireless Sensor Networks. In *Proceedings of the EWSN'04*, 2004.

[5] 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.

[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] W. Leal and S. Bapat and T. Kwon and P. Wei and A. Arora. Stabilizing Health Monitoring for Wireless Sensor Networks. In *8th Intl Symp on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 395–410, 2006.

[8] W. Leal, S. Bapat, T. Kwon, P. Wei, and A. Arora. Stabilizing health monitoring for wireless sensor networks. Technical Report OSU-CISRC-6/06-TR62, Department of Computer Science and Engineering, The Ohio State University, 2006.

[9] Chowkidar Node Status Webpage. http://exscal.nullcode.org/kansei/chowkidar/nodestatus.php.

[10] University of Southern California, Embedded Systems Laboratory. Tutornet: A Tiered Wireless Sensor Network Testbed. http://enl.usc.edu/projects/tutornet/index.html.

[11] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh. Overview of the ORBIT Radio Grid Testbed for Evaluation of Next-Generation Wireless Network Protocols. In *IEEE Wireless Communications and Networking Conference (WCNC)*, 2005.

[12] 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.

[13] 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.