Declarative Failure Recovery for Sensor Networks

Ramakrishna Gummadi *
Nupur Kothari

University of Southern California
{gummadi,nkothari}@usc.edu

Todd Millstein

University of California, Los Angeles
todd@cs.ucla.edu

Ramesh Govindan

University of Southern California
ramesh@usc.edu

Abstract

Wireless sensor networks consist of a system of distributed sensors embedded in the physical world, and promise to allow observation of previously unobservable phenomena. Since they are exposed to unpredictable environments, sensor-network applications must handle a wide variety of faults: software errors, node and link failures, and network partitions. The code to manually detect and recover from faults crosscuts the entire application, is tedious to implement correctly and efficiently, and is fragile in the face of program modifications. We investigate language support for modularly managing faults. Our insight is that such support can be naturally provided as an extension to existing "macroprogramming" systems for sensor networks. In such a system, a programmer describes a sensor network application as a centralized program; a compiler then produces equivalent node-level programs. We describe a simple checkpoint API for macroprograms, which can be automatically implemented in a distributed fashion across the network. We also describe declarative annotations that allow programmers to specify checkpointing strategies at a higher level of abstraction. We have implemented our approach in the Kairos macroprogramming system. Experiments show it to improve application availability by an order of magnitude and incur low messaging overhead.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*; D.3.2 [**Programming Languages**]: Language Classification—*Specialized application languages*; D.4.5 [**Operating Systems**]: Reliability—*Checkpoint/restart* **General Terms:** Wireless Sensor Networks, Macroprogramming, Node-level programming, Failure, Recovery, Checkpointing, Declarative Recovery **Keywords:** WSN, Macroprogramming, Node-level programming, Declarative Failure Recovery, Checkpointing

1. Introduction

Wireless sensor networks consist of a system of distributed sensors embedded in the physical world. Sensor networks promise to al-

low observation of previously unobservable phenomena. They have found use in a variety of domains ranging from scientific experiments involving habitat, environment, and marine monitoring [40] to industrial and civil engineering deployments for measuring device and structure responses [4] to military and commercial applications involving detection and tracking of objects and phenomena [16]. Building practical and reliable sensor network systems is a significant challenge. Sensor networks combine many of the difficulties of traditional embedded systems, including scale, severe resource constraints and an unpredictable operating environment, with the difficulties of traditional distributed systems, including the need for proper synchronization among nodes and the need for fault tolerance.

We focus on the issue of fault tolerance for sensor networks. Maintaining application accuracy and availability in the face of faults is a nontrivial proposition. Software bugs can render a node partially or wholly unresponsive. Network and hardware dynamics such as node failures, burst losses on links, network partitions, and reconfiguration events involving node addition and deletion can completely disable nodes or alter their program state.

For example, consider a vehicle tracking application, in which a group of nodes cooperatively and iteratively refines their estimate of the current position of a moving vehicle. If one or more nodes should fail in the middle of a computation, the resulting estimate can be incorrect because only partial data from the operational nodes is used. Depending on the extent and location of failure, the application may not even be able to form an estimate, effectively rendering it unavailable. Such failures are far more likely in sensor networks, where a large number of nodes are exposed to an unpredictable environment, than in traditional distributed systems.

Researchers have made impressive strides in providing programming platforms (e.g., [13, 18]) and services (e.g., [23, 27]) that simplify the development of sensor network systems. However, to our knowledge, none of these systems provides special support for managing faults. Instead, the programmer must manually implement a failure recovery strategy that is appropriate for the application at hand. In our vehicle tracking example above, a programmer might insert code to track the dependencies among program variables and nodes within the algorithm, detect when a node has failed, and discard failed dependencies in the final output in order to maintain program correctness and availability.

The need for such *ad hoc* recovery code significantly complicates the development of robust sensor-network systems. Recovery code crosscuts the entire application and is intimately tangled with the application logic, making the system difficult to modify and maintain. Further, the recovery code is tedious to implement correctly, for example requiring synchronization among the nodes in the network while maintaining energy efficiency.

We aim to provide declarative support for modularizing the failure concern, allowing sensor-network programmers to easily and reliably identify and recover from faults. Our insight is that this can

be achieved by extending existing *macroprogramming* systems for sensor networks [17, 31]. Unlike the traditional approach, in which programmers directly implement the programs to be run on individual nodes in the sensor network, macroprogramming makes it possible to write a *centralized* program to express a computation. The compiler then automatically produces node-level programs that implement the specified behavior in a distributed manner. Macroprogramming allows programmers to focus on the algorithmic aspects of their applications, without worrying about low-level details like the protocol for communication among nodes.

In this paper, we make the following contributions:

- We describe a simple API for *checkpointing*, a generic recovery approach for a broad class of common sensor-network failures, in the context of a macroprogramming language. The API leverages the macroprogram's centralized view to allow programmers to naturally specify application state to be checkpointed at desired points in the program. The programmer can later roll back to a previously created checkpoint in order to consistently undo the effects of failed nodes, and re-execute the rolled back code with only the set of available nodes. This API is implemented by a novel low-cost distributed algorithm for checkpoint and rollback. The API also supports an important variant of recovery that is designed to preserve application work done during a network partition event, called *Partition Recovery*.

- Our generic recovery API described above is a distinct improvement over *ad hoc* recovery techniques used in traditional sensor network systems, but it still requires the programmer to explicitly interleave recovery logic with the macroprogram. Our second contribution leverages the recovery API to support a form of automated recovery that we call *Declarative Recovery*. Declarative Recovery allows a programmer to provide modular code annotations that specify where checkpoints should be taken, and the macroprogramming system then automatically detects faults and rolls back execution appropriately. It also includes an algorithm to automatically determine at run time the nearest checkpoint to which it is sufficient to roll back in order for recovery to succeed.

- Finally, we push automated recovery even further, to explore a form of *Transparent Recovery*. In this recovery scheme, the system additionally automatically determines where checkpoints should be taken. We describe a simple set of heuristics for placing checkpoints that appropriately handles common macroprogramming patterns.

We have instantiated this approach to failure recovery in sensor networks as an extension of our macroprogramming system called Kairos [17]. We have implemented three qualitatively different sensor network applications using Kairos—localization, target tracking, and data aggregation—and have used them to evaluate the recovery API and the declarative recovery technique. Our primary metrics are the benefits of improvement in correctness and availability of a recovered application in comparison to an unrecovered application, and the performance costs of messaging and memory overheads. Our recovery strategies can improve application availability by an order of magnitude: in some cases, an application is unavailable for 30 times fewer reporting intervals than one which does not incorporate our recovery mechanisms. Our strategies fully preserve application accuracy for two common kinds of faults—software faults and network partitions, incur acceptable messaging overhead (less than 15% for vehicle tracking), and incur about a factor of two additional data memory for storing checkpoints.

To our knowledge, ours is the first work to explicitly address generic failure recovery methodologies for sensor networks. Techniques for detecting and concealing faults and for recovering from failures have been extensively considered in the distributed systems literature (e.g., [8, 14, 29, 39]). Such work, however, has not examined the kind of high-level recovery API and automated recovery techniques that we describe. We are able to support these techniques in a practical manner by leveraging the centralized view of a distributed computation provided by macroprogramming systems.

The rest of the paper is structured as follows. Section 2 motivates the need for failure recovery support in wireless sensor networks, and describes the complexity of manually implementing recovery within node-level programs. Section 3 provides an overview of Kairos, and describes our recovery API on top of Kairos and how it can be used for manual recovery. In Section 4, we describe how we can provide support for declarative and transparent recovery mechanisms. Section 5 details our evaluation of these recovery techniques for several classes of sensor network applications. We describe related work in Section 6. Section 7 concludes and discusses future work.

2. Motivation

Real-world sensor network deployments see significant failures. Figure 1 shows the distribution of failure durations in a real-world sensor network deployment at the James Reserve in Southern California [1]. In this deployment, each sensor periodically sends readings to a base station; failure to receive any readings from a sensor corresponds to a failure of the sending node or one or more other sensors nodes that would otherwise have forwarded the sender's data to the base station. The figure plots the duration of *outages* (intervals during which no data was received from a sensor) for a total of twenty sensor nodes over a period of several months in 2003 and 2004. During this period, each node transmitted data for a total of at least six months. There were a total of 543 outage events during this period.
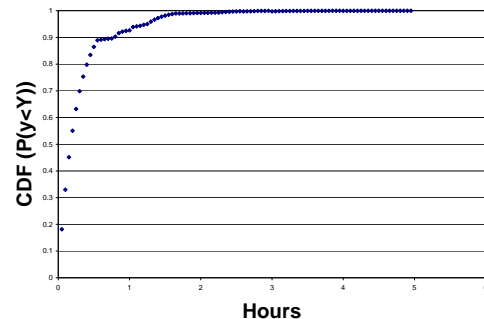

Figure 1—Distribution of outage durations in a real sensor network.

The Cumulative Distribution Function of the duration of outages converges slowly, and outages range from a few minutes to well beyond six hours, with most outages shorter than three hours. Thus, in a real-world sensor network deployment, applications are likely to see a range of node failure and recovery time-scales; there is no single time-scale that one can engineer for. As such, it is desirable for an application to incorporate mechanisms that allow it to function for short periods of time with a smaller set of nodes than it started with, and to re-use nodes that might have been down for extended periods. Such mechanisms can improve the quality of a sensor network computation.

One possibility, then, is for an application writer to manually program failure recovery in sensor network applications. To illustrate some of the problems that arise from manual failure recovery, consider an application in which sensor nodes periodically send both temperature and light readings to a designated base station node, which we assume to be the node with the lowest ID. The

base station aggregates the data it receives in some fashion. Even in this simple scenario, failures must be considered carefully:

1. What should be done if a node fails in some period when the base station has only been able to obtain one of the two sensor values (temperature and light) from the node? For our example, we assume that the base station must remove the effect of the incomplete sensor reading from the aggregation.

2. What should be done if the base station fails? In that case, a new base station must be elected, by finding the live node with the lowest ID. Further, whenever an old base station comes back up, sensor data from the old and current base stations must be merged, and the node with the lower ID must become the new base station.

In a language like nesC [13], the default node-level programming language for the Berkeley sensor motes [2], this application would typically be written as a collection of components, each pertaining to a different task, such as aggregation, leader election, and base-station merging. Every node has the code for all of the components and executes the appropriate procedures from these components depending on its state (*i.e.,* whether it is a normal node, a current base station, or a rebooted old base station).

For ease of presentation we focus on the functionality for data aggregation. Figure 2 shows pseudocode for the two main procedures. The `aggregate_send` procedure is invoked by every node and periodically sends temperature and light readings to the base station. The value of `bs` is set by the leader-election component, which is not shown.

The `aggregate_receive` procedure is invoked by the base station, in order to handle the receipt and aggregation of data from the nodes in the network. In each period (or *epoch*), the base station obtains a list of what it believes to be the live nodes, via a call to the local procedure `get_available_nodes()` (line 15). This list is maintained by the leader-election component (not shown) in an efficient way through a simple membership management protocol. This protocol would be part of the leader election component, whereby every node periodically announces its liveness. The base station then uses a `select()` facility (line 21) to wait for temperature or light data from these nodes (sent via `aggregate_send`) and update local state appropriately. This process repeats until either all expected data from the live nodes has been received (line 25) or a timeout is received, indicating the end of the epoch (line 26).

The `aggregate_receive` procedure handles node failures through checkpoint and rollback, a standard failure recovery approach. The base station takes a checkpoint of its local state at the beginning of each epoch (line 13). When a timeout is signaled, indicating that some live nodes did not provide both sensor values, the base station restores this checkpoint (line 27). This has the effect of removing all data obtained in the current epoch from the aggregation, thereby ensuring consistency. (It is possible to perform finer-grained recovery, for example retaining sensor readings in the current epoch from any node for which both values were able to be obtained. However, doing this would require the programmer to manually track dependencies to ensure consistency, which is tedious and error prone.)

To handle base station failures, we assume that whenever a node determines that the base station has not broadcast its liveness, as part of membership management described above, that node triggers leader election. To handle the situation when an old base station comes back, the current base station checks the live nodes at each epoch for a node with a lower ID, invoking the merge functionality if required (lines 16–17).

Manual recovery as illustrated by our example has a number of drawbacks:

```
  node bs;
  //executed at every sender
  void aggregate_send() {
1: uint temp,light;
2  for(;;) {
3:  sleep(SAMPLE_INTERVAL);
4:  sample(temp);
5:  sample(light);
6:  send_sample(temp,bs);
7:  send_sample(light,bs);
  }
 }

  Ckpt ckpt;
  //executed at base station
  void aggregate_receive() {
8: time next_epoch;
9: list node_list, received_list;
10:uint av_l, av_t, count, timeout;
11:boolean done;
12:for (;;) {
13: ckpt=take_local_ckpt();
14: next_epoch=get_cur_time()+SAMPLE_INTERVAL;
15: node_list=get_available_nodes(),received_list=NULL;
    //check if node_list has an old base station
16: if (hasLower(node_list,id())) {
17:  ...//invoke merge()...
    }
18: timeout=SAMPLE_INTERVAL;
19: done=FALSE;
20: while (!done) {
    //wait till timeout or at least one node sends
21:  received_list=select(TEMP_T|LIGHT_T,node_list,&timeout);
22:  if(received_list!=NULL) {
23:   //read temp and/or lt values; compute averages...
24:   //remove node from node_list if bs got temp,lt...
25:   if (node_list==NULL) done=TRUE;
    }
26:  else{//bs timed out=>nodes in node_list are dead
     //restore node-local state to previous epoch
27:   restore_local_ckpt(ckpt);
    }
   }
28: sleep(next_epoch-get_cur_time());
  }
 }
```

Figure 2—Send and receive procedures for data aggregation in a node-level program with manual recovery.

1. The code for the recovery concern is tangled with the rest of the application logic. For example, the base station must explicitly check for the presence of an old base station after accessing the live nodes (line 16) and must explicitly restore a taken checkpoint upon detecting a failure in the middle of an epoch (line 27). Further, because a checkpoint could be restored at any point in its dynamic lifetime, managing checkpoints is nonmodular. For example, if the inner `while` loop in Figure 2 were defined in its own function, the checkpoint `ckpt` would have to be restored from there, requiring it to either be a global variable (whose deletion would then need to be manually managed to save space) or to be explicitly passed to the function.

2. Proper recovery may require manual tracking of dependencies across nodes. In our example, only the base station's local state is of interest upon a node failure, so local checkpoint and recovery (`take_local_ckpt` and `restore_local_ckpt`) are sufficient. However, suppose the base station's local state had dependencies with local state at other nodes in the network. In that case, whenever the base station required a rollback, the failed dependencies at other nodes would also have to be

tracked and removed to maintain consistency. Further, whenever these dependencies change, through program maintenance or extension, the recovery code must likewise be updated.

Dealing with network partitions also makes dependency tracking harder because a partition causes some nodes to be disconnected from others, thereby causing their states to drift as nodes in the two partitions work independently. After a partition is repaired, one option is for the programmer to simply discard the work done by nodes from one half of the partitioned network. However, this is sub-optimal, because work done by both sets of nodes can be integrated into the long-term state of the healed network, which improves the quality of the final results. But, again, this requires careful tracking of dependencies between data across nodes during and after the partition has occurred.

3. Similarly, proper recovery may require synchronization across nodes. If dependencies exist across nodes, requiring rollbacks at multiple nodes, the programmer must be sure to synchronize these rollbacks to ensure consistency. Otherwise, one node could restart its rolled-back execution before another node has been fully rolled back. Manual synchronization is difficult to implement both correctly and in an energy-efficient manner, which is critical on today's resource-constrained sensor nodes.

3. Generic Checkpoint Recovery in a Macroprogramming System

We first describe the particular macroprogramming language and system we use throughout this paper, called Kairos [17]. While we have concretely examined and evaluated the techniques described in this paper within Kairos, we believe that the key concepts can be adapted to other macroprogramming languages like Regiment [42].

3.1 An Overview of KAIROS

In this section, we briefly review the Kairos macroprogramming system; it is described in more detail elsewhere [17].

Kairos lets a programmer directly express the desired *global behavior* of a distributed computation. The programmer achieves this by writing a centralized program in which sensor network data can be manipulated as ordinary program variables. The Kairos compiler then translates the centralized program into programs that execute on individual nodes, with the support of the Kairos runtime.

We summarize the Kairos language abstractions here. Kairos augments a host language with a small number of new programming primitives, which allow a distributed computation on a sensor network to be expressed centrally. The programming model is analogous to that of mainstream imperative programming languages: Kairos has a sequential semantics by default and a centralized memory model. As such, Kairos fits well as an extension to commonly used languages. We have built a Kairos extension to Python, which we use in our implementation and experiments reported here. The detailed description of our compilation and runtime techniques is available in [17].

Kairos decouples a sensor network program from the underlying node topology, thereby making it instantiable on an arbitrary topology. The `node` data type is an abstraction of a network node. Nodes can be conveniently manipulated using a `nodelist` iterator data type that presents a set-based abstraction of a node collection. Kairos makes sure that the values contained in these variables are visible consistently and efficiently at all nodes. The function `get_available_nodes()` provides access to the `nodelist` representing all nodes in the network, while the `get_neighbors(node)` function returns the current list of `node`'s radio neighbors. Given the broadcast nature of wireless communication, a neighbor list is a natural abstraction to build interacting groups of nodes in a program, and is similar to *regions* [42] and *hoods* [43].

Kairos provides a natural way to access the program state at any node from within the centralized program. A *node-local variable* is a program variable that is instantiated per node. A particular node's version of a variable can be accessed by the macroprogram through a `var@node` syntax. All other variables are instantiated only once within the network, and are called `central` variables. Kairos respects the scoping, lifetime, and access rules of variables imposed by the host language.

```
  void av() {
1: nodelist full_node_set;
2: node iter, bs;
3: uint sleep_interval=1000;
4: uint nodelocal count=1, av_t=0, av_l=0;
5: uint nodelocal sensor temp, lt;
6: full_node_set=get_available_nodes();
7: bs=get_first(sort(full_node_set));
8: for (;;) {
9:   sleep(sleep_interval);
10: for (iter=get_first(full_node_set);iter!=NULL;
         iter=get_next(full_node_set)) {
11:   av_t@bs=(av_t@bs*(count@bs-1)+temp@iter)/count@bs;
12:   av_l@bs=(av_l@bs*(count@bs-1)+lt@iter)/count@bs++;
    }
  }
 }
```

Figure 3—Example macroprogram for computing average temperature and light readings.

Figure 3 shows the Kairos code that uses these abstractions for continuously computing the sample averages for light and temperature readings and storing them at a base station node. It works as follows. In lines 1, 2, and 3, we declare variables to represent the list of nodes in the network, a temporary node, a node that will be chosen as the base station, and the time to sleep between averaging intervals. In lines 4 and 5, we declare node-local variables. `temp` and `lt` are special node-local variables (indicated by their `sensor` attribute) that are continuously updated with new readings from a node. For each of the node-local variables, a copy of the variable with the same name exists at each node in the network. In line 6, we store the list of active nodes in the network in the variable `full_node_set`, and in line 7 we instantiate the node with the lowest id as the base station node. Finally, in lines 8–12, we cause the network the repeatedly fetch temperature and light samples and store their average at the base station.

Kairos implements the distributed version of a macroprogram in a *network-efficient* manner. The Kairos runtime has a distributed caching layer that makes sure updates to central variables are visible across the network consistently. The caching layer also buffers updates from node-local variables so that the programmer can perform synchronous reads and writes. For example, in lines 11–12 of Figure 3, the macroprogram updates the base station's node-local variables in sequence, while in a node-level program such as Figure 2, the programmer is responsible for managing network messages that may arrive any time and out of order. Kairos minimizes communication overhead for both data and control through three techniques: by allowing asynchronous execution at nodes and minimizing their control flow synchronization; by exploiting relaxed data consistency semantics where possible in order to further reduce control traffic overhead; and by caching remote variables for reads and filtering unnecessary writes [17].

3.2 Recovery in Macroprograms

In addition to checkpointing, which was discussed in Section 2, there are two approaches for programming recovery into macroprograms. The simplest approach is for runtime support to provide

error notifications, leaving it to the programmer to manually deal with failures. For example, the runtime can return an error when reads to a remote variable fail. Thus, in Figure 3, accesses to node-local variables `av_t`, `av_l`, `count`, `temp` and `lt` in lines 11–12 can return a special error code when a node is unavailable. But such a facility only solves one of the three problems with node-level manual recovery described in Section 2. While the programmer is relieved of the burden of manually synchronizing such accesses across nodes because the macroprogram's runtime implements such synchronization, the programmer must still deal with the first two problems: she would have to add checks around each access to a node-local variable (there would be five such checks in lines 11–12, for example), and manually track dependencies across such node-local states.

The second approach is to augment the language with a transaction facility. While such a facility could potentially solve all the three problems of manual recovery, it would be heavy-weight unless carefully implemented. Support for nested transactions would be necessary in order to minimize lost work during recovery, but such transactions are difficult to implement efficiently and correctly in a distributed setting because of their potential for causing deadlocks and livelocks [21]. Thus, we use checkpoint-based recovery.

3.3 Manual Failure Recovery for Macroprogramming

In this section, we examine a checkpointing approach to manual failure recovery in the context of macroprogramming. In particular, we describe a small checkpointing API for Kairos. The key novelty is the way in which this API leverages Kairos' centralized view of the network: programmers specify checkpoints at the granularity of the macroprogram, and the runtime system carefully ensures the corresponding node-level programs take consistent checkpoints and rollback in a synchronized manner when a failure is detected. Programmers are still responsible for manually managing checkpoints, thereby suffering from some of the drawbacks described in Section 2. These drawbacks are addressed by our automated recovery strategies, which build on the checkpointing API and are described in the next section.

The Checkpointing API

In a Kairos macroprogram, the programmer may call the following function at any point:

```
Ckpt take_ckpt(nodelist nl);
```

This function takes a consistent checkpoint at every node in the specified node list. By a consistent checkpoint, we mean that no node in nodelist `nl` proceeds in the computation until it knows that all other nodes in `nl` have also taken the checkpoint. This call returns a handle to the checkpoint.

To rollback to a checkpoint, a programmer may call the following function:

```
boolean restore_ckpt(Ckpt ckpt);
```

This function takes a previously created checkpoint as an argument and restores the state at each node (again, consistently) to that at the specified checkpoint. Execution of the restored program then resumes at the statement following the point where the specified checkpoint was taken.

Figure 4 shows how this API can be used to implement a version of fault tolerant sensor averaging in Kairos. It meets the requirements described at the beginning of Section 2 except for merging data from old base stations, which we describe in the following subsection. The recovery code that is additional to the macroprogram in Figure 3 is shown in bold. The basic idea behind the recovery code is to use two checkpoints for the two failure scenarios that

```
   Ckpt ckpt1, ckpt2;
   void av() {
1: nodelist full_node_set;
2: node iter, bs;
3: uint sleep_interval=1000;
4: uint nodelocal count=1, av_l=0, av_t=0;temp;
5: full_node_set=get_available_nodes();

6: ckpt1=take_ckpt(full_node_set);
   //Check if we have to take another checkpoint
7: if (ckpt1.restored){
8:  full_node_set=get_available_nodes();
9:  ckpt1=take_ckpt(full_node_set)
   }

10:bs=get_first(sort(full_node_set));
11:for (;;) {
12: sleep(sleep_interval);

13: ckpt2=take_ckpt(bs);
14: full_node_set=get_available_nodes();

15: for (iter=get_first(full_node_set);iter!=NULL;
          iter=get_next(full_node_set)) {
16:   av_t@bs=(av_t@bs*(count@bs-1)+temp@iter)/count@bs;
17:   av_l@bs=(av_l@bs*(count@bs-1)+lt@iter)/count@bs++;
   }

18: if (_failed) {
19:  full_node_set=get_available_nodes();
20:  if (member(bs,full_node_set)) {
21:   //bs still alive=>another node crashed
22:   restore_ckpt(ckpt2);
23:  } else {
24:   restore_ckpt(ckpt1);
   }
  }
 }
}
```

Figure 4—Example macroprogram with manual recovery code.

must be recovered from: when a base station crashes, `ckpt1` created in line 6 is used, and when any other node crashes, `ckpt2` created in line 13 is used. Whenever one or more nodes fail during the execution, the runtime ultimately triggers the recovery code in lines 18–24. This is because failures of nodes are detected in the background by the runtime and exposed through an internal variable called `_failed`, which the programmer can check anywhere in the program.

In case a node other than the base station crashes, the programmer restores checkpoint `ckpt2` in lines 20–22, and in case the base station itself crashes, the programmer restores checkpoint `ckpt1` in line 24. If `ckpt1` is restored, execution resumes at line 7. The programmer takes another checkpoint in lines 7–9 if `ckpt1` has been restored. If `ckpt2` is restored, execution resumes at line 14. Thus, as long as any node other than `bs` crashes, the state at `bs` is unaffected, because of its recovery via checkpoint `ckpt2`. Further, since restoring a checkpoint reinstates a previous state of the program, the actual values of node-local variables that a program uses in the time between taking and restoring a checkpoint is immaterial. Lines 16–17 of Figure 4 exploit this property by not checking for the return values of node-local variables. Our implementation currently returns a well-defined error code, which is useful if a programmer wants to implement finer-grained recovery.

A macroprogram written to use the checkpointing API is said to use *checkpoint-rollback recovery* (CRR). CRR solves the last two problems of node-level recovery described in Section 2 as follows:

1. Unlike the manual checkpoints taken in the pseudocode in Figure 2, which are local to a particular node, the Kairos checkpointing API provides globally consistent checkpoints. Therefore, the programmer is relieved from manually tracking dependencies across nodes: as long as all nodes that have dependencies among one another are checkpointed and the call to `take_ckpt` is placed at a globally consistent point, all dependencies will be properly handled automatically.

2. The Kairos runtime automatically synchronizes nodes when a checkpoint is taken or restored. For example, a node is only allowed to resume execution after restoring its local checkpoint once all other nodes have also restored their checkpoints. Therefore, the programmer is completely relieved of the burden of node synchronization.

Detecting Faults

Fault detection is a significant research challenge in its own right. In this paper, we assume non-malicious faults and use a simple, yet practical, fault detection strategy. A fault is said to occur when a read or write to a node fails after three successive retries. Variable reads and writes use a simple request/response protocol in Kairos. This protocol has a three-second timeout, a reasonable upper-bound on real-world latencies in sensor networks. When a fault is detected, the `_failed` flag is set.

Implementing Checkpoints

Kairos implements the checkpointing API, and its various components, in its runtime. This task involves coordinating the relevant node-level runtimes to efficiently take and restore checkpoints.

By the time a program invokes `take_ckpt(nl)`, the runtime ensures that the value of `nl` is available at every node in the network. Each node within `nl` takes its own local checkpoint, sends out a completion message, and stops the node-level program execution until it hears the same message from other nodes. Nodes that are not in `nl` need not actually take a local checkpoint, but they still have to participate in a global consensus algorithm [11]. In the general case, this synchronization can be expensive, requiring $O(N^2)$ reliable point-to-point transmissions for a network of size N.

We propose two novel optimizations for communication reduction. First, we exploit the broadcast nature of wireless sensor networks in order not to require every node to communicate with every other node. We build consensus using the following algorithm, which has two phases of execution. In the first phase, a node reliably broadcasts "Done/Wait" to its immediate neighbors after it takes its local checkpoint. Whenever it hears "Done/Wait" from all neighboring nodes, it enters the second phase of execution by reliably broadcasting "Done" in the local domain. Once it hears "Done" from all current neighbors, the node independently determines that a consistent global checkpoint has been taken. Intuitively, this algorithm works because a node would not have entered the second phase of the protocol if any neighbor had not yet completed or even entered its own first phase.

The cost of this protocol is clearly at most $2N$ reliable local broadcasts. Another distinguishing feature of this protocol is that nodes only need to synchronize during this operation but otherwise execute completely asynchronously with respect to one other.

Second, we optimize a common case scenario in which a checkpoint is repeatedly taken over a single node. For example, in line 13 of Figure 4, we repeatedly checkpoint state at `bs`. In such cases, the runtime can avoid global synchronization by only taking a local checkpoint. The runtime implements the correct consistency semantics so that such a checkpoint is valid. Our local checkpointing implementation uses the `libckpt` library to save the private process state (the data, heap and stack segments) locally at a node.

When a program invokes `restore_ckpt(ckpt)`, the runtime restores the program state from the local checkpoints of nodes stored in `ckpt` structure. The distributed component of this operation uses the same machinery involved in `take_ckpt()`, except that the remaining nodes should obviously not expect protocol messages from the failed nodes. After the live nodes have agreed to consistently restore a `ckpt`, each node's runtime locally restores its own state.

Summary

The checkpointing API provides an abstraction that specifies recovery actions at the level of the macroprogram itself. The Kairos runtime carefully ensures that consistent checkpoints are taken at each local node. This checkpointing mechanism is conceptually similar to other distributed checkpointing techniques, all of which, including ours, are variants of Chandy and Lamport's algorithm [9]. The main novelty is that our implementation is asynchronous and optimized for the locally communicating and broadcast nature of sensor networks. Moreover, we are not aware of similar language-level recovery techniques that are tightly integrated with the underlying distributed programming system, a feature which is useful in providing support for partition recovery, as described in the next section.

3.4 Recovering from Partitions

Checkpointing can lose work between the last checkpoint and when a `restore_ckpt()` is called. If faults affect a single node, invoking CRR is the right choice if we only want to ensure consistency of the macroprogram state. In the case of many continuous-output applications, such as vehicle tracking, it also happens to be the optimal choice because it causes the macroprogram to respond both rapidly and correctly to network dynamics, and compute the continuous output with no loss of accuracy.

However, CRR on its own is not sufficient to properly handle network partitions. We define a partitioning as an event which causes one or more live nodes to be disconnected from the rest of the network. Suppose a network partition occurs anywhere between lines 11–15 in Figure 4. With CRR, Kairos would rollback the computation and resume it independently on both halves of the partition. There are now two `bs` values, each of which keeps accumulating averages. When the partition heals, we need a mechanism to let a programmer specify how to unify the work done by each side of the partition.

We provide such a mechanism, called *Partition Recovery*, which combines the global work done by each group during the partition. The goal is to preserve the values of variables representing the long-lived program state, such as `av_l@bs`, `av_t@bs`, and `count@bs`. The programmer invokes partition recovery by specifying a *merge* function along with the macroprogram. The runtime indicates that the partition has healed by setting a `_healed` variable, and the programmer can detect this condition similar to the check used for node failures in line 15 of Figure 4.

In order to make the ensuing discussion clear, we show the entire code for the macroprogram in Figure 5. It is augmented with the code for dealing with partitions in bold.

Figure 5 works as follows. When a partition occurs, the test for `_failed` in line 18 would succeed. One half of the partition that contains the base station would work with a fewer set of nodes because its runtime restores `ckpt2` to line 14 of the macroprogram, while the other half that does not contain the current base station will additionally obtain a new base station when its runtime restores `ckpt1` to line 7 instead. Thus, there would be two global runtimes, each of which is the union of the local runtimes of its constituent nodes. We note that the original base station does not lose its long term state (*i.e.,* its values of `av_l`, `av_t`, and `count`) because the

```
  Ckpt ckpt1, ckpt2;

  node bs, bs_P1, bs_P2;
  uint nodelocal count=1, av_l=0, av_t=0;

  void av() {
1: nodelist full_node_set;
2: node iter;
3: uint sleep_interval=1000;
4: uint nodelocal temp;
5: full_node_set=get_available_nodes();
6: ckpt1=take_ckpt(full_node_set);
   //Check if we have to take another checkpoint
7: if (ckpt1.restored){
8:  full_node_set=get_available_nodes();
9:  ckpt1=take_ckpt(full_node_set)
   }
10:bs=get_first(sort(full_node_set));
11:for (;;) {
12: sleep(sleep_interval);
13: ckpt2=take_ckpt(bs);
14: full_node_set=get_available_nodes();
15: for (iter=get_first(full_node_set);iter!=NULL;
        iter=get_next(full_node_set)) {
16:  av_t@bs=(av_t@bs*(count@bs-1)+temp@iter)/count@bs;
17:  av_l@bs=(av_l@bs*(count@bs-1)+lt@iter)/count@bs++;
   }
18: if (_failed) {
19:  full_node_set=get_available_nodes();
20:  if (member(bs,full_node_set)) {
21:   //bs still alive=>another node has crashed
22:   restore_ckpt(ckpt2);
23:  } else {
24:   restore_ckpt(ckpt1);
    }
   }

25: if (_healed) {
26:  merge_av();
    }
   }

  }
 }

 void merge_av()
 {
27:bs=min(bs_P1,bs_P2);
28:av_l@bs=(av_l@bs_P1*count@bs_P1+av_l@bs_P2
         *count@bs_P2)/(count@bs_P1+count@bs_P2);
29:av_t@bs=(av_t@bs_P1*count@bs_P1+av_t@bs_P2
         *count@bs_P2)/(count@bs_P1+count@bs_P2);
30:count@bs=count@bs_P1+count@bs_P2;
 }
```

Figure 5—Example macroprogram for recovering from partitions.

rollback of its partition is only until line 14. As long as they are separate, both partitions work independently thereafter.

Later, when the two partitions merge, the runtimes of the two halves will independently detect this condition, because the two distributed runtimes maintain information about which nodes are available. Before the merge, each runtime maintains its own copy of the central variables, and the copies may become out of sync. After the merge, the programmer may require access to both copies, in order to determine an appropriate value to use for that central variable upon program resumption. A programmer can indicate a central variable var whose values from the two partitions should be saved by declaring two additional central variables var_P1 and var_P2. For example, in the beginning of Figure 5, the programmer declares bs_P1 and bs_P2. Just before the two runtimes of a parti-

tion merge, bs_P1 and bs_P2 are updated respectively with values from each of the two partitions.

The merge function is invoked by the programmer separately for each runtime in line 26, after she detects that the underlying partition has healed, by testing for the _healed flag in line 25. The merge function synchronizes the two runtimes by making the first caller wait until the second caller has also invoked merge_av. merge_av first updates the global variable bs of the macroprogram to the lower-valued base station. It then updates the av_l, av_t, and count values at bs. When it exits, the two runtimes are considered unified, and the application resumes execution at line 11.

One observation we can make regarding the application domain of sensor networks is that it is often possible to write merge functions that follow a well-known idiom. Figure 6 describes some common example applications and suitable merge functions.

State Type	Example	Merge Function
Aggregatable scalars	Sum, average, count	Simple aggregation
Linearly combinable vectors and matrices	Vector aggregates, auto- and cross-correlations, covariance, Fourier transforms	Textbook compositional formulae
Non-aggregatable scalars	Max, min, quantiles, histograms, quantiles, etc.	Duplicate insensitive counting/sketch theory, q-digests, approx. aggregates, etc.
Spatiotemporal state	Isobars, contours, etc.	Model/problem-specific but simple low-state spatiotemporal interpolated composition

Figure 6—Common tasks and their merge functions.

4. Automated Recovery Strategies

While our checkpointing API for macroprogramming is a significant step from node-local manual recovery, it still requires the programmer to manually create checkpoints, manage their lifetimes explicitly, and restore to the appropriate checkpoint at necessary places within the application logic.

4.1 Declarative Recovery Annotations

In order to relieve the programmer from dealing with such issues, and in order to only allow her to reason about recovery modularly, we have designed a *Declarative Recovery (DR)* annotation technique. This annotation takes the following form: <nodelist, merge_func> where nodelist is an expression that evaluates to a list of nodes potentially affected by a fault, and the optional merge_func specifies a merge function to be used after a partition. The nodelist argument is specified using a set-theoretic notation, with support for basic operations of union, intersection, and difference. Such an annotation may be placed at any line in the macroprogram, and more than one such annotation may be present in a given macroprogram.

When a programmer places an annotation <nodelist, merge_func> at some point in the program, she is indicating that the global program state is consistent at that point, and therefore that this is an appropriate point at which to take a checkpoint. When such an annotation is encountered during execution, the runtime automatically takes a checkpoint at all nodes in nodelist (using the checkpointing API described in the previous section). The runtime also starts a new *recovery scope* and watches for any failed or merged nodes in the background. This recovery scope lasts for the dynamic extent of the annotation's smallest enclosing program block.

When any remote access encounters a failure within a recovery scope, the runtime *automatically* rolls back the computation at each node in the macroprogram to the most recent relevant checkpoint. Relevance is determined by the nodelist argument to an annotation, which indicates that forward progress can be made from this

point as long as at least one node in `nodelist` is live. Given this information, the runtime can automatically rollback to the checkpoint that discards the least amount of work while ensuring forward progress. We describe this rollback algorithm in more detail in the next subsection. Furthermore, when new nodes are added to the system or when a partition heals, the runtime also rolls back the computation, applies the specified merge function, and resumes the computation.

```
  void av() {
1: nodelist full_node_set;
2: node iter, bs;
3: uint sleep_interval=1000;
4: uint nodelocal count=1, av_t=0, av_l=0;
5: uint nodelocal sensor temp, lt;
6: full_node_set=get_available_nodes();

7: <full_node_set,NULL>
8: full_node_set=get_available_nodes();

9: bs=get_first(sort(full_node_set));
10:for (;;) {

11: <{bs},NULL>
12: full_node_set=get_available_nodes();

13: sleep(sleep_interval);
14: for (iter=get_first(full_node_set);iter!=NULL;
         iter=get_next(full_node_set)){
15:   av_t@bs=(av_t@bs*(count@bs-1)+temp@iter)/
             count@bs;
16:   av_l@bs=(av_l@bs*(count@bs-1)+lt@iter)/
             count@bs++;
    }
   }
  }
```

Figure 7—Example macroprogram to illustrate Declarative Recovery (DR).

Figure 7 shows our averages example augmented with recovery annotations (lines 7 and 11). Lines 8 and 12 are additional code for ensuring that the program can make progress without the failed nodes, and are executed immediately after the macroprogram has been rolled back to the corresponding points. For simplicity, we do not show a merge function, which would be very similar to the one in Figure 5; thus, the second arguments of the annotations in lines 7 and 11 are `NULL`. In line 7, we annotate `full_node_list` as the set of nodes over which the checkpoint is defined. In line 11, we activate another recovery scope, defined only over the base station. This annotation indicates that forward progress can be made from line 11 as long as the base station is still live. Therefore, whenever any node other than `bs` fails during the annotation's recovery scope, the runtime rolls the program back only to line 12, and re-initializes the set of currently available nodes. If the base station fails, however, the runtime instead rolls back to line 8, and subsequently chooses a new base station in line 9.

These declarative recovery annotations eliminate the problems of manual checkpointing described earlier. A simple annotation tells the runtime where checkpoints should be taken. The runtime then automatically creates and manages these checkpoints, detects failures and determines an appropriate checkpoint to restore, even across function boundaries. In this way, the recovery code is much more insulated from the application logic and much more robust to application updates. Finally, the runtime also automatically garbage collects checkpoints as they become inactive, as described in the next subsection.

4.2 Selecting and Managing Checkpoints

In a macroprogram with several annotations, the checkpoints created by active annotations can be dynamically managed as a single list. In this list, a checkpoint A follows a checkpoint B if the line of code at which A was taken is executed after the line at which B was taken at run time. The runtime continuously tracks nodes' membership status in the background to discover if one or more nodes have failed or been partitioned. If it detects such a condition, it searches this checkpoint list for a checkpoint with an annotation that has specified at least one live node. Intuitively, the programmer intends each checkpoint to represent both a globally consistent state, and, orthogonally, a liveness condition that declares that the macroprogram can make forward progress if execution is retried from that point on, after discarding the effects of failed nodes during checkpoint recovery.

The runtime allocates and maintains memory for checkpoints in an efficient and distributed manner. Metadata associated with a checkpoint, which includes the list of nodes over which the checkpoint was taken and the checkpoint's parent checkpoint in the list, is replicated at every node. One valuable optimization is that, if this metadata does not change when the next checkpoint is taken at the same place, global synchronization is averted. Thus, in Figure 7, when the runtime repeatedly takes a checkpoint over `bs`'s state, it can avoid global communication and synchronization after the first time. This is because of our observation that if two checkpoints are taken over the same node list, the older checkpoint can be safely replaced by the newer checkpoint—the older checkpoint will never be used in favor of the later checkpoint because (a) our liveness requirement during rollback applies equally to both checkpoints, and (b) our requirement to minimize work lost will cause the later checkpoint to preferentially be chosen over the older one.

The runtime reclaims storage allocated for checkpoints in the following simple fashion. Whenever execution encounters an annotation, the runtime takes a checkpoint at that location, and then discards any previously taken checkpoints at that code location. This strategy lazily discards checkpoints; an alternative would have been for the Kairos compiler to carefully discard checkpoints whenever execution exits the static program scope in which the annotation is declared, but our approach requires less work on the part of the compiler, and is comparably efficient. Furthermore, before restoring the state corresponding to a selected checkpoint, the runtime discards the saved memory associated with all later checkpoints in the list.

4.3 Transparent Recovery

While Declarative Recovery significantly simplifies programming recovery, it still requires the programmer to annotate code. In addition to identifying points in the code where consistent checkpoints can be taken, the programmer has to indicate what the minimal set of live nodes is at such points in order to optimize lost work. For example, in Figure 7, the second annotation's first argument must contain the base station in order to avoid losing `bs` state. An interesting question arises whether it is possible to provide completely *transparent* recovery, without programmer involvement at all.

We have taken a first step in this direction using a simple heuristic that we call Transparent Recovery (TR). In Transparent Recovery, the need for supplying declarative annotations is eliminated, but the programmer must still supply a merge function for the program, because merge functions are inherently application-specific. We only allow one merge function to be provided; it is declared with a special attribute indicating that it is the merge function.

Transparent Recovery works as follows. The Kairos compiler generates code to take a checkpoint after each update to a variable of type `node` or `nodelist`. Thus, going back to the *original* example (Figure 3), the compiler would direct the runtime to take a

checkpoint of `full_node_set` after line 6, of `bs` after line 7, and of `iter` after line 10.

Transparent recovery can be sub-optimal, losing more work than necessary, because it is not possible to infer the nodes that must be live at a given point to ensure that forward progress can be made. Therefore, TR's rollback strategy must be conservative: upon the failure of a node n, TR rolls back to the latest checkpoint C such that *C and all earlier checkpoints* do not include node n. Intuitively, it is safe to roll back to C if this condition is met, since nothing in the program execution up to C depended upon node n. If no such checkpoint exists, the runtime system simply rolls back to the beginning of the macroprogram.

Because all nodes are checkpointed at line 6 in Figure 3, this code represents a worst case of sorts for TR; failures always cause rollback to the beginning of the macroprogram. However, our experiments in Section 5 illustrate that the technique is practical for other common sensor-network applications. For example, TR is appropriate for many continuous-output applications like vehicle tracking, because nodes in such a network do not accumulate long-term state.

5. Evaluation

In this section, we describe the results of experiments conducted on a wireless testbed using an implementation of Kairos and the recovery mechanisms described in this paper. We quantify the efficacy of our recovery techniques along various dimensions: error in application quality, application availability, and messaging and memory overhead.

5.1 Methodology

Implementation: We implemented Kairos, and the recovery techniques for Kairos partly in Python (using its embedding and extending APIs) and partly in C. The Kairos runtime uses EmStar [12] to implement end-to-end reliable routing and topology management. Our Kairos implementation runs on 32-bit embedded platforms such as the Stargate [3], as well on PCs. We have presented the details of our Kairos implementation in [17]. To this implementation, we added the recovery API described in Section 3, and the compiler and runtime support for Declarative Recovery (DR) and Transparent Recovery (TR) described in Section 4. All experiments reported in this paper use this implementation.

Applications: We evaluate the efficacy of recovery in Kairos using three representative sensornet applications written in Kairos, the complete code for which is given in [17]. The three applications are: vehicle tracking, for which an explicitly distributed algorithm based on Bayesian belief propagation is given in Liu *et al.* [24]; node localization in a sensor network, for which we macroprogram the distributed algorithm based on cooperative multi-lateration as given in Savvides *et al.* [35]; and quantile estimation, for which we macroprogram the distributed algorithm, based on the concept of a summarizing data structure called q-digests, as given in Shrivastava *et al.* [37].

These applications place different demands on Kairos, yet, as we show below, Kairos is able to satisfactorily recover each application. Vehicle tracking is an instance of a locally-communicating, continuously-sensing, latency sensitive, periodic (duty-cycling) application. Localization is an example of a globally-communicating, single-shot, latency insensitive application, driven by network events such as node addition, deletion, mobility, and reconfiguration. Finally, q-digest is a network-wide locally-communicating application, whose output and latency sensitivity requirements depend on its use: for collecting statistics over a continuously changing sensor field, it can be configured to be a latency tolerant, continuous output application; however, for low frequency rare event monitoring, it can be configured as a latency sensitive single-shot application.
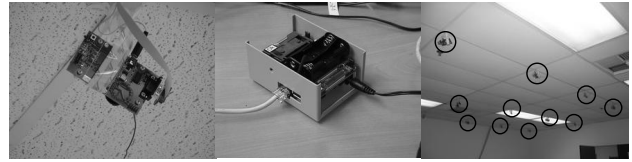

Figure 8—A single Mica-Z controlled by a PC (left), a single Mica-Z attached to a Stargate (center), and Mica-Z's (circled) on the ceiling (right).

The Testbed: Our testbed consists of 36 nodes, of which 15 are Stargates with an attached Mica-Z mote (Figure 8). The remaining 21 are emulated nodes, each node being emulated by one EmStar process. Each emulated node uses a real (not emulated) Mica-Z mote for all communication. These Mica-Z motes are mounted on the ceiling of our laboratory (Figure 8). This setup allows us to simulate real-world multihop configurations without being constrained by the limited memory resources of the current generation of motes.

In our testbed, all nodes are within a single physical hop of each other, but we configure nodes to multi-hop through other nodes in order to more closely mimic real deployments. Specifically, we arrange these node to form a 6x6 2D torus topology.

Experimental Setup: A single run of an experiment measures application performance metrics (described below) for N faults, where N ranges from 0 to 15. We inject three types of faults into the system, and in a given run, all injected faults are of the same type. In a software fault (SW), the application instance at a node is killed, leaving the Kairos runtime operational. In this case, for example, remote reads of raw (unprocessed) sensor data can be satisfied by the Kairos runtime. In a hardware fault (HW), the entire node is stopped, so that neither the Kairos runtime nor the application can send or receive messages. When injecting a software or hardware fault, we are careful to keep the network itself connected. Finally, we also inject a network partition (PR), where N nodes are partitioned from the system, and the partition then heals after 2 minutes. In all cases, the network is started with no faults, and faults are injected immediately after the first call to *get_available_nodes*▷◁ has succeeded.

We set algorithm parameters as follows. For vehicle tracking, we assume a constant speed target moving randomly within the 6x6 grid. Other parameters of the algorithm in [24] are scaled to fit our topological dimensions. For localization, coordinates of beacon nodes are randomly perturbed with Gaussian noise according to the parameters in [35]. The q-digest application uses a Kairos application to construct the routing tree along which the data digests are sent. We configure q-digest to periodically (every 100s) send digests.

In all experiments, the recovery latency for checkpointing, after failure detection, was less than a minute.

Comparing Recovery Strategies: We evaluate transparent recovery (TR) for software faults (TR-SW) and for hardware faults (TR-HW). In the programs we evaluate, TR-SW and TR-HW are respectively equivalent to DR-SW and DR-HW because they happen to roll back to the same checkpoint in each case, and are thus exhibit identical performance. For evaluating the efficacy of recovery after partition healing, we evaluate Declarative Recovery with non-null merge functions in the annotations (DR-PR). Since Transparent Recovery is primarily meant for applications that don't accumulate long-lived state, we do not evaluate TR-PR. For DR-PR, we have inserted declarative annotations into each of the Kairos applications; none of our applications requires more than 5 annota-

tions, although the largest of our applications is nearly 250 lines of macroprogramming code, which is roughly how much large macro-programmed tasks are in practice.

We compare these strategies against two baseline cases: the performance of the application without any faults (NF), and the performance of the application without recovery (NR) in the presence of faults.

Metrics: Our comparison is based on the following four quantitative metrics. Our first metric is application *availability*, defined for two of our applications, vehicle tracking and q-digest. In these two applications, the application periodically (say every T-second intervals) returns a result (the current location of the vehicle, or the current median). When a fault occurs, the application may or may not be able to return an answer at a given instance. Define U_F to be the fraction of intervals during which an unrecovered application (NR) did *not* return an answer. Define U_R analogously, but for an application with a recovery strategy applied. Then, we define application availability to be $\log_{10}\!\triangleright\!\frac{U_F}{U_R}\!\triangleleft$, a metric that is commonly used for representing availability. This *logarithmic* metric defines applications which return an answer during 0.999 of the intervals to have $1x$ (or 100% more) availability compared to one which returns an answer during 0.99 of the intervals.

Our second metric measures application *error*. This metric applies to all three applications, of course, and is defined for vehicle tracking as $\frac{|z_N - z_R|}{|z_N|}$, z_N is the approximation computed by NF, and z_R is the approximation computed by either TR-SW, TR-HW or DR-PR. The metric is similarly defined for our other applications. Our last two metrics measure the messaging and memory overhead of recovery. They are defined as the additional fraction of messages sent, or memory used, relative to NF.

We chose these metrics because, for recovery techniques to be practical in realistic multi-hop scenarios, they need to be lightweight in addition to being expressive.

5.2 Results

In Figures 9, we plot the availability of the vehicle tracking application as a function of the number of faults. Notice that the advantages of recovery are apparent even with one failure; the increase in availability is more than $1x$, indicating that recovery strategies *reduce the number of intervals during which an answer is available by a factor of 10*. As the number of faults increases, this factor rises to nearly 30 ($10^{1.5}$). The availability for the q-digest application is qualitatively similar (Figure 10). For both applications, different strategies exhibit slightly different availabilities, mostly due to differences in the latency of recovery across the three approaches.
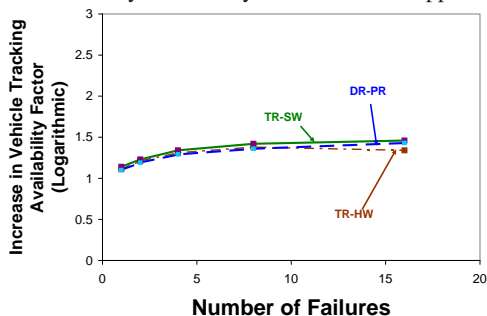

Figure 9—Availability comparison of TR-Software (TR-SW), TR-Hardware (TR-HW), and DR-Partition Recovery (DR-PR) strategies with increasing node failures.

In Figure 11, we plot the relative error in the position estimate as a function of the number of faults for the vehicle tracking application. Our baseline for comparing our recovery strategies is the
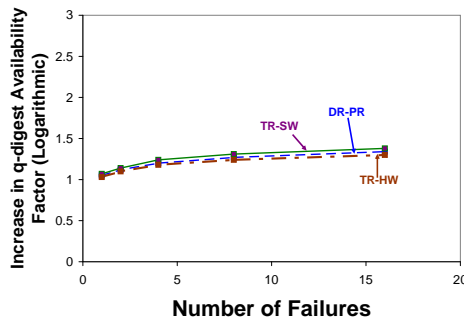

Figure 10—Availability comparison of TR-Software (TR-SW), TR-Hardware (TR-HW), and DR-Partition Recovery (DR-PR) strategies with increasing node failures.
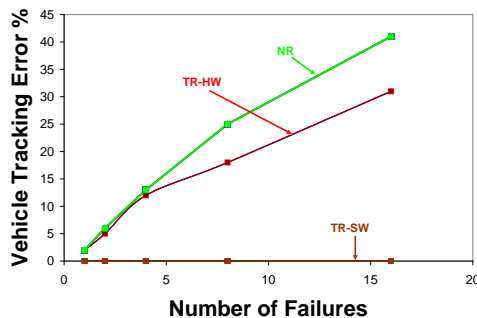

Figure 11—Accuracy comparison of TR-Software (TR-SW), TR-Hardware (TR-HW), and No Recovery (NR) strategies with increasing node failures.

case where no recovery strategy is employed (NR). The relative error under NR is indicative of the loss of application fidelity inherent in failure; for example, when nodes fail, a vehicle tracking application is essentially left with a less dense network than before, adversely affecting tracking quality. Our main observation in this graph is that, while the application accuracy degrades linearly with increasing numbers of faults (TR-HW), this degradation is no worse than the relative error under NR. This indicates that the loss of application accuracy is entirely inherent in node failure, and recovery does not exacerbate this loss. On the contrary, recovery *reduces* application error: TR-HW has lower application error than NR, because the latter has lower availability resulting in missed readings and therefore a more erroneous track (since the tracking algorithm uses a smoothed history of position readings). Furthermore, note that TR-SW exhibits no relative error at all; when the Kairos runtime is able to respond with sensor readings, even if the application instance at the node itself is dead, the overall application is still able to preserve fidelity. Finally, we do not show DR-PR in this graph; partition healing is not relevant to an application in which the answer (the position estimate) is continuously changing.

In Figures 12 and 13, we plot the relative error of the various recovery strategies for q-digest and localization. The interesting difference between this graph and Figure 11 is that, as expected, DR-PR also exhibits zero application error since partition recovery is able to recover lost work by merging two q-digests, or location estimates. Furthermore, since medians and location estimates are relatively less sensitive to node loss than vehicle tracking, the magnitude of error for TR-HW is lower. Even in this case, however, this error is comparable to the application error without fault recovery (we do not shown application error under NR for localization because a single failure causes the application to not successfully terminate).
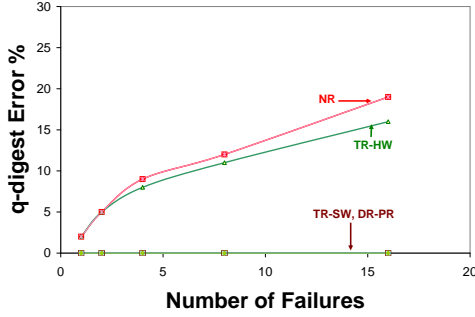
Figure 12—Accuracy comparison of TR-Software (TR-SW), TR-Hardware (TR-HW), DR-Partition Recovery (DR-PR), and No Recovery (NR) strategies with increasing node failures.
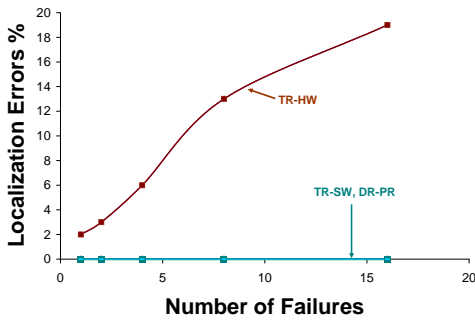

Figure 13—Accuracy Comparison of TR-Software (TR-SW), TR-Hardware (TR-HW), and DR-Partition Recovery (DR-PR) strategies with increasing node failures.

In Figure 14, we show the messaging overheads for the various recovery strategies for each of the applications. The error bars in the figure depict the variation in overhead with the number of faults. We see that communication overhead of these mechanisms is independent of the severity of faults, and depends mostly on the nature of the application. TR-SW and TR-HW are almost equal because they share the same logic when invoked, and incur no more than 25% additional messaging overhead. DR-PR incurs twice as much overhead as TR-HW or TR-SW for some applications like q-digest that span the entire network, and, therefore, involve a large number of nodes. For applications like vehicle tracking in which, at any given time, only nodes within a certain locality are involved, the overhead of DR-PR is almost a constant and quite small (about 15%).
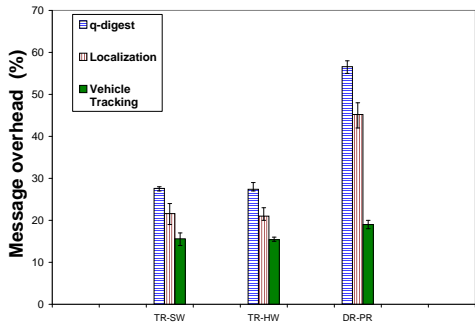

Figure 14—Message overhead comparison of TR-SW, TR-HW, and DR-PR strategies.

Finally, we see in Figure 15, that TR requires between 2.2–2.5 times the data memory of an application without recovery, which measures the amount of checkpointing state maintained. Today's sensor nodes have different program, SRAM, and flash memories. Since SRAM (data) memory can be stored inside a (much larger) flash, this problem is less severe. Nevertheless, clearly, this is an aspect of our system that could benefit from some optimization. This memory overhead is independent of the number of faults. It does depend only on application characteristics, specifically the average nesting depth of checkpoints. Interestingly, for our applications, this nesting depth happens to be comparable (slightly more than 2 for an average execution trial), hence the memory overhead appears to be the same across applications.
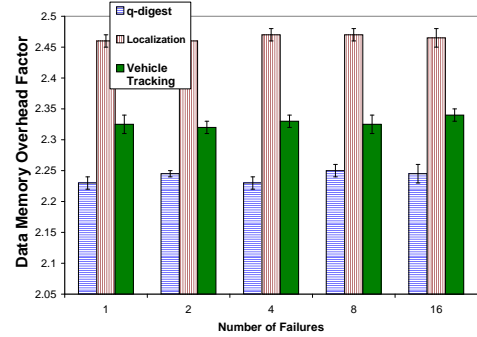

Figure 15—Memory overhead for CRR.

Summary. Our recovery strategies can improve application availability by an order of magnitude, while preserving application accuracy for certain kinds of faults (software faults, and network partitions). They incur acceptable messaging overhead (less than 15% for vehicle tracking), and a factor of two additional data memory for checkpointing. While TR works well for continuous output applications like vehicle tracking, those requiring longevity, such as q-digest, benefit from having declarative annotations with merge functions for partition recovery.

6. Related Work

There are two primary approaches for generic rollback-based recovery schemes, a survey of which is given in [11]. Such schemes can be classified as either checkpoint-based or log-based. Log-based protocols tend to have unpredictable message logging requirements, which are hard to provision for in memory-restricted sensor nodes. However, log-based protocols are predominantly used in databases [28] and file systems [34] because they have access to a large and persistent disk storage. CRR is checkpoint-based, and there is an extensive set of algorithms and implementations of distributed checkpoint schemes in a variety of domains ranging from loosely coupled message passing systems to tightly coupled multiprocessors [5, 6, 10, 19, 22, 30, 32, 36, 38]. A taxonomy and survey of such schemes is given in [20]. Two important features of our checkpointing API are that (a) it is easier to use than most of these techniques because it leverages the macroprogramming abstraction, and (b) it is implemented efficiently over the broadcast facility in wireless sensor networks.

Declarative recovery through annotations is a novel aspect of our work. We are not aware of prior work similar to these language-level constructs, even though a growing body of literature exists for augmenting systems such as MPI with recovery APIs and libraries [7]. Also, recently, there is a renewed interest in implementing systems components using declarative approaches [15, 25], but they do not directly deal with recovery.

Our partition recovery support is also novel. Others have proposed application-specific merge functions in varied contexts such as distributed file systems [33] and mobile computing [41]. However, such systems have not been widely popular mainly because their generality means merge functions are hard to write. We believe that it is simpler to write merge functions for sensor network applications because they are frequently numerical in nature. Madden *et al.* have proposed a form of merge functions for query processing in sensor networks [26], but those merge functions were for normal processing inside SQL queries, and not for recovery. Finally, we are not aware of any prior work that considered recovery in sensor networks, either in the context of macroprogramming systems such as [31, 42, 43] or otherwise.

7. Conclusions and Future Work

Failures are a critical concern for sensor-network systems, and one that crosscuts entire applications. In this paper, we have described the problems for manual failure detection and recovery in sensor networks, and we show how the notion of *macroprogramming* can be used to largely untangle the failure concern from the application logic. First, we have designed a generic checkpointing API for macroprogramming systems that leverages the centralized view of a network to allow checkpoint and rollback to be specified at natural points in the overall application. Second, we explored two automated recovery strategies, which significantly raise the level of abstraction for specifying recovery and serve to further insulate the recovery concern from the rest of the application. We have implemented our checkpointing API and automated recovery strategies in the Kairos macroprogramming system, and experimental results illustrate their utility and practicality.

Several avenues for future work remain. First, it would be useful to gather more experience with our techniques on real-world deployments. Second, our work on transparent recovery is only a first step; we plan to examine a range of applications to better understand appropriate heuristics for transparent recovery that will be widely applicable. Finally, our recovery implementation is relatively unoptimized; in future work we will use program-analysis techniques to automatically minimize work lost in recovery and to minimize the memory overhead of checkpoints.

References

[1] James reserve. URL {http://www.jamesreserve.edu/}.

[2] Micaz mpr2400. URL {http://www.xbow.com/Products/productsdetails.aspx?sid=101}.

[3] Stargate platform. URL {http://www.xbow.com/Products/XScale.htm}.

[4] R. Adler, P. Buonadonna et al. Design and deployment of industrial sensor networks: Experiences from the north sea and a semiconductor plant. *SenSys 2005*.

[5] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Trans. Softw. Eng.*, 24(2):149–159, 1998. ISSN 0098-5589.

[6] R. Baldoni, F. Quaglia, and B. Ciciani. A vp-accordant checkpointing protocol preventing useless checkpoints. In *SRDS '98*, page 61. IEEE Computer Society, 1998. ISBN 0-8186-9218-9.

[7] G. Bosilca, A. Bouteiller et al. Mpich-v: toward a scalable fault tolerant mpi for volatile nodes. In *SC'02*, pages 1–18. IEEE Computer Society Press, 2002.

[8] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. *OSDI, 2004*.

[9] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985. ISSN 0734-2071.

[10] E. Elnozahy. *Manetho: fault tolerance in distributed systems using rollback-recovery and process replication*. PhD thesis, 1994.

[11] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv., 2002*.

[12] J. Elson, S. Bien et al. Emstar: An environment for developing wireless embedded systems software. *CENS-TR-9, 2003*.

[13] D. Gay, P. Levis et al. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *PLDI 2003*.

[14] J. Gray. Why do computers stop and what can be done about it? *SRDS'86*.

[15] T. Griffin and J. Sobrinho. Metarouting. *SIGCOMM 2005*.

[16] L. Gu, D. Jia et al. Lightweight detection and classification for wireless sensor networks in realistic environments. *SenSys 2005*.

[17] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. *DCOSS 2005*.

[18] J. Hill, R. Szewczyk et al. System architecture directions for network sensors. *ASPLOS, 2000*.

[19] D. B. Johnson. *Distributed system fault tolerance using message logging and checkpointing*. PhD thesis, 1990.

[20] S. Kalaiselvi and V. Rajaraman. A survey of checkpointing algorithms for parallel and distributed computers. *Sadhana, Vol. 25, Part 5*.

[21] E. Knapp. Deadlock detection in distributed databases. *ACM Comput. Surv.*, 19(4):303–328, 1987. ISSN 0360-0300.

[22] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Softw. Eng.*, 13(1):23–31, 1987. ISSN 0098-5589.

[23] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire tinyos applications. In *Sensys, 2003*.

[24] J. Liu, J. Reich, and F. Zhao. Collaborative in-network processing for target tracking. *EURASIP, 2002*.

[25] B. Loo, J. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: Extensible routing with declarative queries. *SIGCOMM 2005*.

[26] S. Madden, M. J. Franklin, J. Hellerstein, and W. Hong. TAG: A tiny AGgregation service for ad-hoc sensor networks. *OSDI, 2002*.

[27] M. Maroti, B. K. G. Simon, and A. Ledeczi. The flooding time synchronization protocol. *Sensys, 2004*.

[28] J. E. B. Moss. Log-based recovery for nested transactions. In *VLDB '87*, pages 427–432, 1987. ISBN 0-934613-46-X.

[29] B. Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. *QREI, 1995*.

[30] R. H. B. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):165–169, 1995. ISSN 1045-9219.

[31] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. *DMSN, 2004*.

[32] J. S. Plank. *Efficient checkpointing on MIMD architectures*. PhD thesis, Princeton, NJ, USA, 1993.

[33] G. J. Popek, R. G. Guy, J. Thomas W. Page, and J. S. Heidemann. Replication in ficus distributed file systems. *Workshop on Management of Replicated Data, 1990*.

[34] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52. ISSN 0734-2071.

[35] A. Savvides, C. Han, and S. Srivastava. Dynamic Fine-Grained localization in Ad-Hoc networks of sensors. *MOBICOM, 2001*.

[36] R. D. Schlichting and F. B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, 1983. ISSN 0734-2071.

[37] N. Shrivastava, C. Buragohain, S. Suri, and D. Agrawal. Medians and beyond: New aggregation techniques for sensor networks. *SenSys'04*.

[38] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985. ISSN 0734-2071.

[39] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. *FTCS, 1991*.

[40] R. Szewczyk, A. Mainwaring, J. Polastre, and D. Culler. An analysis of a large scale habitat monitoring. *Sensys, 2004*.

[41] D. Terry, M. Theimer et al. Managing update conflicts in bayou, a weakly connected replicated storage system. *SOSP 1995*.

[42] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. *NSDI, 2004*.

[43] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. *MobiSys, 2004*.