Failure Recovery in a Macroprogrammed Sensor Network System

Ramakrishna Gummadi* Omprakash Gnawali* Ramesh Govindan* Todd Millstein†

ABSTRACT

Sensor network macroprogramming systems like Kairos can greatly simplify programming a distributed sensor network application by expressing it as a centralized sequential computation on shared node state. However, the accuracy and timely availability of the results can be adversely affected by software, node hardware, and network failures. Ideally, a macroprogramming system should relieve the programmer from having to explicitly handle such failures by providing high-level facilities for failure recovery. In this paper, we show that there exist *generic* techniques that can recover sensor network computations from various classes of failures. We then evaluate the efficacy of two recovery invocation strategies: *directed* recovery, in which Kairos invokes a declaratively-specified generic recovery method; and *transparent* recovery in which Kairos itself invokes an appropriate recovery method at a suitable point in the macroprogram, without programmer assistance. We evaluate these techniques on the primary metrics of application accuracy and availability against three qualitatively different sensor network applications, written in Kairos, running on a 36-node test-bed. Our main result is that transparent recovery can ensure worst-case correctness of the recovered application to within 40% of that of the non-faulting application across a wide range and severity of software and hardware faults, and can improve an application's availability in the presence of faults by a factor of up to 2.75.

1. INTRODUCTION

Research in wireless sensor networks (WSNs) has made impressive strides in providing infrastructural tools and services [18, 21] with the overall goal of simplifying the development of robust and scalable distributed applications [27]. Existing programming systems take two forms: language-level mechanisms like nesC [9] and associated component-oriented OS-level platforms like TinyOS [12] that support programming at the granularity of a single node; and databases-inspired higher-level SQL-style programming frameworks like TinyDB [20] and Cougar [8], layered on top of these node-level interfaces, that then permit declarative access to an entire collection of sensor nodes.

More recently, the sensor networks community has focused on a class of programming methodologies that go beyond declarative, SQL-restricted models. Using these methodologies, it is possible to write a single *centralized* program to express a distributed computation running on a sensor network. This approach is called *macroprogramming* [23, 11].

In prior work, we have developed *Kairos* [11], a sensor network macroprogramming system (Section 2). Kairos presents an abstraction of a sensor network as a collection of nodes whose memory is centrally accessible, and which can be tasked collectively from within a *single* sequential program. In Kairos, programmers *implicitly* express both distributed data flow and distributed control flow; this top-down approach contrasts favorably against bottom-up ad hoc distributed programming models, which require the programmer to implement detailed data path and control flow interactions among nodes using explicit messages. Given the single centralized program, Kairos' compile-time and runtime systems automatically synthesize and enforce the necessary low-level distributed interactions.

Kairos, or any other macroprogramming system for that matter, raises the level of abstraction to the point where the programmer need not be aware of detailed network and node behavior. In such systems, however, maintaining application correctness and accuracy in the face of faults—software faults and network dynamics—is an important challenge. Software bugs can violate application invariants or render unresponsive the compiled macroprogram at some nodes. Network dynamics such as node failures, burst losses on links, network partitions, node addition or reconfiguration, or uncaught network packet errors, can introduce errors in the results produced by a macroprogrammed application. For example, consider a vehicle tracking application [16] in which a group of nodes cooperatively and iteratively refine 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. Depending on the extent and location of failure, the application may not even be able to form an estimate, effectively rendering it un-

*University of Southern California. {gummadi, gnawali, ramesh}@usc.edu
†University of California, Los Angeles. todd@cs.ucla.edu

available.

In this paper, we explore the space of recovery mechanisms and strategies for macroprogramming systems in the context of Kairos. A key challenge is the search for sufficiently *generic* approaches that are usable in a variety of programs and failure modes. Our first contribution is the design of two low-overhead generic recovery mechanisms. The first mechanism, *Opportunistic State Reconstruction*, improves the availability of Kairos' runtime state in the face of transient failures by reconstructing the state of neighboring nodes based on overheard message transmissions. When a neighbor fails, a node can respond on its behalf, allowing an application to make progress. The second technique, *Checkpoint-Rollback-Recovery* improves macroprogram accuracy across a broad class of failures by *checkpointing* application state at those code points in the macroprogram that enclose program actions involving a vulnerable set of nodes. If the node set changes anywhere within a given scope, the macroprogram can *rollback* to the most recent saved state and resume the computation. We also study an important variant of checkpointing that is designed to preserve the application work done during a network partition event, called *Partition Recovery*.

Our second contribution leverages these generic recovery mechanisms to support a form of *automated recovery*, which significantly raises the level of abstraction for programmer-specified recovery policies, as opposed to *manual recovery* strategies, which require the programmer to explicitly interleave recovery logic with the macroprogram. In particular, we investigate *directed recovery*, which allows a programmer to provide structured declarative code annotation constructs of the form: ⟨`failure_type`, `recovery_args`⟩. Using such annotations, programmers can easily specify where recovery is required and for what kinds of failures. For example, a programmer can succinctly annotate macroprogram code fragments that should be protected from transient faults. When the Kairos runtime detects the specified failure type, it automatically invokes an appropriate generic recovery mechanism. For the purposes of failure detection, we identify broad classes of faults typically encountered in sensor networks, as well as heuristics for classifying such faults into application-visible failures.

Finally, our third contribution explores a stronger form of automated recovery that we call *transparent recovery*. We have designed a simple yet effective program analysis technique by which the Kairos compiler can automatically determine where a particular recovery mechanism is appropriate, thereby dispensing with the need for programmer annotations. For the purposes of recovery via rollback, transparent recovery 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.

We have implemented three qualitatively different sensor network applications using Kairos—localization, target tracking, and data aggregation—and have used them to evaluate both manual and automated generic recovery strategies, and the three recovery mechanisms mentioned before. Our metrics are correctness of a recovered application in comparison to the non-faulting application, application availability, recovery latency, performance (memory and network) overhead, and implementation complexity.

The main evaluation result is that transparent recovery can ensure worst-case application correctness to within 40% of the non-faulting application across a wide range and severity of software and hardware faults. It can also improve an application's availability in the presence of faults by a factor of up to 2.75. This performance comes at 10% higher message overhead compared to manual recovery, which provides no improvement in application accuracy or availability in the presence of two common classes of faults. We also find that a programmer can define simple global invariant assertions, using the directed recovery strategy, to overcome certain complex logical or hardware bugs that severely impact the output of the macroprogram. Our failure classification heuristic has low ($< 5.1\%$) misclassification, while our recovery mechanisms perform competitively on component-level metrics like recovery latency and message cost ($< 40\%$ and $< 20\%$ respectively higher than manual recovery in the worst case, but several times lower in some cases). The only consistent downside to transparent recovery is the memory overhead ($\sim 2.5\times$), but, given the historical progress in memory densities and sizes, we do not believe that memory will be a serious bottleneck in future WSNs. Finally, we provide simple guidelines to help a sophisticated application programmer wield precise control over which recovery mechanism to use depending on application characteristics.

To our knowledge, ours is the first work to explicitly address generic failure recovery methodologies in macroprogrammed WSNs. We describe simple automated generic recovery mechanisms as well as techniques for specifying and inferring recovery policies in the form of recovery strategies, but there is clearly more to be done. For example, we have not considered sophisticated forms of manual recovery nor real-world deployments and topologies in our evaluation. Techniques for detecting and concealing faults, and for recovering from failures have been extensively considered in the distributed systems literature [10, 26, 22, 4]. Such work has, however, not examined programmable or automated recovery techniques. We are able to support these techniques in Kairos by leveraging its centralized view of the distributed computation.

2. AN OVERVIEW OF KAIROS

In this section, we briefly describe the Kairos macroprogramming system [11]. Kairos lets a programmer directly express the desired *global behavior* of a distributed computation. As a motivating example, consider a distributed computation that builds a shortest path tree rooted at a node **N** in a sensornet. A global specification for an algorithm to compute this tree can be stated as:

For each node **n** in the network, its *parent* is that neighbor whose distance to **N** is shortest.

Kairos allows the above algorithm to be expressed succinctly as a centralized program. The Kairos compiler then translates the centralized program into programs that execute on individual nodes, with the support of the Kairos runtime. In our example, the resulting local program might, for instance, cause a node to repeatedly poll its neighbors about their current distance to **N**, process the received distances, and pick that neighbor whose distance to **N** was smallest.

2.1 Kairos Abstractions

To achieve this goal, Kairos provides four *programming primitives* which can be incorporated into most programming languages, and which capture the necessary distributed control flow and data consistency semantics. The programming model consists of a sequential tasking model and a centralized memory model that supports a synchronous read and write view of sensor node state. Intuitively, such a computing model simplifies sensornet programming because a distributed computation like a Dijkstra Shortest Path algorithm can be programmed by directly expressing standard centralized "textbook" algorithms in the source language. This is an explicit design goal of Kairos.

Kairos' first primitive decouples the program from the underlying topology, thereby making it instantiable on an arbitrary topology. The `node` datatype is an abstraction of a network node and behaves as an ordinal datatype, permitting various operations on its instances, like equality, ordering, and type testing. Nodes can be conveniently manipulated using a `node_list` iterator data type that presents a set-based abstraction of a node collection.

The second primitive provides a shared memory abstraction for nodes. A *node-local variable* is a variable that is instantiated per node. A particular node's version of the variable can be accessed centrally through the `variable@node` syntax. Kairos does not impose any restrictions on how node-local variables may be manipulated. However, Kairos' compiler respects the scoping, lifetime, and access rules of variables as imposed by the host language. Kairos also supports *central variables*, for which there exists exactly one programmer-visible instantiation. These variables are useful for global coordination (e.g., booleans and loop-control variables) and for global lists of nodes. By default, variables in Kairos are node-local; central variables are denoted using the `central` keyword.

The third primitive, the `get_neighbors()@node` function, returns the current list of the `node`'s radio neighbors. Given the broadcast nature of wireless communication, this is a natural abstraction for sensor network programming, and is similar to *regions* [28] and *hoods* [29].

The fourth primitive is a `time_queue` abstraction that implements a temporally ordered series of named actions. Every action item in the queue has a `time` member element, and these actions are dispatched from the queue no earlier than `time`, but in a best-effort manner. Insertions need not be in temporal order, and an application can demultiplex on the type of the fired action to centrally determine what to do. Applications use this abstraction to ensure temporal consistency of a sensor network computation.

2.2 Kairos Architecture

Given a centralized macroprogram using the above constructs, it is straightforward to produce a *non-optimized* distributed version. The Kairos compiler simply identifies and translates references to remote data into calls to the Kairos *runtime*. Control flow synchronization is added for every basic block, a single copy of the centralized variables is kept in the network, and each node only executes actions in the block around which its node identifier is used.

Such a transformed binary is then linked to the Kairos runtime, and can be distributed to all nodes in the sensor network through some form of code distribution and node re-programming facility [19, 13]. When a copy is instantiated and run on each sensor node, the Kairos runtime exports and manages program variables that are owned by the current node but are referenced by remote nodes; these objects are called *managed objects* in Figure 1. In addition, it also caches copies of managed objects owned by remote nodes in its *cached objects* pool. Remote accesses to both sets of objects are managed through queues as asynchronous request/reply messages that are carried over a potentially multihop radio network.

The user program that runs on a sensor node can call synchronously into the Kairos runtime for reading remote objects, as well as for accessing local managed objects. These calls are automatically generated by the Kairos compiler. The runtime accesses these objects on behalf of the program after suspending the calling thread. The runtime uses additional background threads to manage object queues, but this aspect is transparent to the application, and the application is only aware of the usual language threading model. Such an architecture for Kairos is depicted in Figure 1.

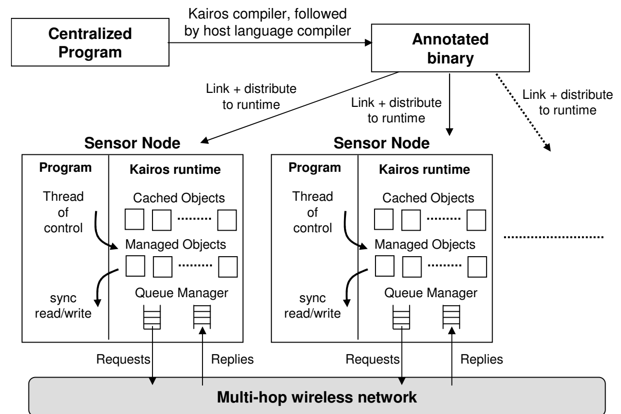

Figure 1: Kairos Programming Architecture

Kairos strives to implement a distributed version of such

a centralized program in a *network-efficient* manner, because battery life is a limiting factor in WSNs, and communication is the chief consumer of energy. Even though a Kairos macroprogram is specified as a sequential program implementing synchronous access to remote variables, Kairos does not require fine-grain node *synchronization* and can therefore be energy-efficient. Sensor network distributed programs like routing tree construction and localization [24] can tolerate eventual consistency, so the Kairos runtime can permit individual node instances to execute without almost any explicit synchronization overhead. Applications like vehicle tracking require temporally consistent, but not necessarily synchronized, access to remote node state. Programmers can use Kairos' `time_queue` to express these consistency requirements, and the run-time can decide to synchronize node execution only when necessary.

2.3 Kairos Example

We now show how the routing tree construction can be expressed in Kairos. The full pseudocode for achieving this is shown in Figure 2, and is similar to the one in [11]. It is mainly an expository aid, but can also be used as a subcomponent of a sophisticated bulk-data routing macroprogram.

```
1: void buildtree (central node root=0)
2:   node parent;
3:   unsigned short dist_from_root, node_list neighbors;
4:   central full_node_set;
5:   central unsigned int sleep_interval=1000;
     //Initialization
6:   full_node_set=get_available_nodes();
7:   for (central node iter=get_first(full_node_set);
          iter!=NULL; iter=get_next(full_node_set))
8:     iter=get_local_node_id();
9:     if (iter==root)
10:      dist_from_root@iter=0; parent@iter=iter;
11:    else dist_from_root@iter=INF;
12:    neighbors@iter=create_node_list(
                         get_neighbors()@iter);
     //Main Tree Construction
13:  full_node_set=get_available_nodes();
14:  for(;;)  //Can be replaced with terminating cond.
15:   sleep(sleep_interval);
16:   for (central node iter1=get_first(full_node_set);
           iter1!=NULL; iter1=get_next(full_node_set))
17:    for (node iter2=get_first(neighbors@iter1);
               iter2!=NULL;
               iter2=get_next(neighbors@iter1))
18:     if (dist_from_root@iter2+1<dist_from_root@iter1)
19:       dist_from_root@iter1=dist_from_root@iter2+1;
20:       parent@iter1=iter2;
```

Figure 2: Procedural Code for Building a Shortest-path Routing Tree

Figure 2 highlights all the essential aspects of Kairos. The `get_available_nodes()` function used in lines 6 and 13 is provided by the Kairos runtime and is used to conceptually iterate over all nodes in the network in the subsequent `for` loops. The loop in lines 7–12 initializes the node-local `dist_from_root` variable at each node. It also initializes each node's `neighbors` variable using the `get_neighbors()@n` function. The loop in lines 13–20 eventually picks a shortest path from a node to the `root` node, by continually updating each node's `parent` to be the closest neighbor. To do so, the node-local variable `dist_from_root` at each node is accessed remotely from its neighboring nodes in lines 18–19.

3. GENERIC RECOVERY TECHNIQUES

In this section, we first consider to what extent recovery can be done manually, what the various automated mechanisms are, and, finally, how to identify and classify faults into program level failures so that an automated recovery mechanism appropriate for that failure class can be used.

3.1 Manual Recovery Mechanisms

We first consider two inter-related questions:

- Where in the macroprogram are software or hardware faults made visible as application failures in Kairos, and to what extent can we hope to minimize their program level visibility and scope by manual programming?
- What *generic techniques* are available to the programmer, and how can they be represented as simple *manual strategies*?

Let us revisit the tree builder code in Figure 2. We note that the program, in the form currently written, is fragile with respect to node and link failures: if a remote node fails while it is being read from its neighbor using its `dist_from_root` variable inside the loop between lines 17–20, the macroprogram would be left read-blocked permanently. It is possible to manually *rewrite* this program to make it more robust by eliminating line 12 of the code, and inserting the following fragment before line 17 to recover from failed links:

```
1: neighbors@iter1=create_node_list(get_neighbors()@iter1);
   //Check if a node's parent still exists or is dead
2: central boolean parent_exists=false;
3: for (node iter2=get_first(neighbors);
        iter2!=NULL && iter2!=root;
        iter2=get_next(neighbors))
4:   if (iter2==parent@iter1) parent_exists=true;
5: if (!parent_exists@iter1) dist_from_root@iter1=INF;
```

The main idea here is to refresh the neighborhood list during every iteration of the top-level $for()\{\}$ loop in line 14, to make sure that a node's `parent` is still alive. Unfortunately, such a rewriting does not fully solve the problem, because a node may still fail in the same manner, although the window of vulnerability is now shortened to the time interval between retrieving the neighborhood list and reading the `dist_from_root` values from each of the nodes within the list (*i.e.,* to lines 17–18 in Figure 2). Furthermore, this code introduces network overhead by forcing each node to retrieve the full list of neighbors during every iteration (it can roughly double the number of messages).

One alternative to rewriting a program as above is to use simple manual recovery strategies, which is what explicitly distributed programs do today. For example, Kairos can return an error on a failed node access, and the programmer can take corrective actions to consistently remove the prior contribution to the global state from that node, and remove the node from future consideration. Thus, one simple generic technique is "discard failed node", representable as a "Discard" manual recovery strategy.

Unfortunately, the task of correctly coding "Discard" becomes complex if the node data has been used to manipulate node state at other nodes (*e.g.,* the node being used as a parent in the routing tree, or as a beacon to localize other nodes). However, it is a low resource overhead procedure, and may be suitable for continuous output applications like vehicle tracking that do not keep or use long-term node state.

Another simple and generic manual repair technique is to include a *continuous repair* generic recovery technique to periodically refresh, and, if necessary, re-establish self-consistent program state (*i.e.,*, values of central variables, and node-local variables). Such a "Redo" manual recovery strategy is useful for long-lived applications like data routing. The `time_queue` abstraction of Kairos helps here, by letting a programmer globally schedule periodic refreshes, and does not require much code: of course, this comes at the cost of being inconsistent within the refresh periods, and at a high refresh cost when there are no failures. We evaluate these two instances of manual recovery strategy in Section 5. Finally, we note that today's distributed programs typically include such recovery logic.

In Figure 2, the chief difficulty in dealing with failures is the lack of an *atomicity* guarantee in accessing data on a node throughout the macroprogram. One approach for supporting generic recovery is then to provide explicit atomic constructs. Unfortunately, providing explicit `atomic` constructs for sensor network macroprograms can be suboptimal. We find that it is natural and commonplace in centrally programmed distributed algorithms to create global state about nodes (such as the `node_list`'s of nodes in a network, or about neighbors at a node in Figure 2) much earlier in the macroprogram than when it is actually used, effectively linking variables separated by large chunks of the macroprogram, and creating long-range atomicity dependencies. Thus, the whole macroprogram is impacted even in the case of a localized failure.

Thus, if we want a reliable and efficient macroprogram, the programmer has to sprinkle elaborate, well-thought-out recovery logic at many places in the macroprogram, with each recovery fragment being only slightly different than the others. Another issue also complicates manual recovery strategies: if we want efficiency and high fidelity, we cannot treat all failures as equal. For example, in vehicle tracking, simply discarding a node permanently just because it is experiencing transient faults may not be acceptable. Instead, we would like to employ a "Retry" manual recovery strategy that encapsulates the familiar "retry" generic recovery technique used for overcoming "soft" errors. This means, for n vulnerable points and m distinct failure types, the programmer must craft nm disjoint recovery fragments scattered throughout the code, in the worst case.

We can make the following observation regarding the nm problem. It is possible to associate a generic automated recovery mechanism with a failure class, as we will see in Section 3.3. Furthermore, a statement in the macroprogram is typically vulnerable to only or two failure classes, depending on what data it references. These data variables can be passed as parameters to the recovery mechanism when Kairos detects failures on those variables during normal execution. This means that, if we decompose recovery actions into an orthogonal set of recovery mechanisms, we can avoid the complexity of considering, for every statement in the macroprogram, all possible failure types and corresponding individualized, unparameterized variable recovery logic.

3.2 Automated Recovery Mechanisms

In this section, we examine recovery mechanisms that provide linearly and hierarchically *scoped* recovery methods based on the generic "retry," "undo," and "redo" techniques, in order to achieve high performance (*i.e.,* low state and work discarding) recovery. The "retry" based mechanism is called *Opportunistic State Reconstruction (OSR)*, and is considered in Section 3.2.1. The "undo" and "redo" based mechanism is called *Checkpoint Rollback Recovery (CRR)*, and is considered in Section 3.2.2.

3.2.1 Opportunistic State Reconstruction

We propose a novel mechanism for capturing the automated recovery principle of "retry". It is based around the simple idea of exploiting the cheap broadcast property of wireless sensor networks to promiscuously store a node's state at neighboring nodes. Then, when the node is temporarily unavailable, a neighboring node can hopefully fulfill the request from its cache. This mechanism imposes low memory overhead because only program variables, but not arbitrary remote memory contents, are potentially cached. It has low network overhead because requests that could not be directly satisfied do not incur any additional cost to be satisfied using OSR.

The mechanism is proactive, with the sender broadcasting the updated value along with the current time as an unacknowledged transmission whenever a local variable is updated. This protocol means that, for node-local variables that are remotely accessed at least once after they are updated, there is typically no overhead; for those variables that are not remotely accessed, there is a penalty of one broadcast, while frequently accessed items can actually gain a small performance advantage.

It is also correct in the following sense: it is based on robust "soft state" principles, with the node-local variables being available at least as long as the node itself is alive.

Central variables are not handled by this method because they need much stronger guarantees using deep replication: an extreme instance is when all nodes carry a copy of all central variables. It is the relatively rare, and, therefore, network cost intensive node-local variables whose availability needs to be improved. The more widely used central variables place different demands on availability and communication performance.

The access protocol using OSR is simple: without OSR, Kairos normally attempts a remote data access at most five times. With OSR, when a Kairos runtime tries to access a remote node's runtime, and times out on the third data access attempt, the accessing Kairos runtime makes up to two additional attempts with a special flag in the packet header indicating that the access can be a cached read. All nodes in the routing path to the destination look at such requests and see if they can satisfy it. The lifetime of a cached variable is determined by the global macroprogram state, and is, thus, well defined. If more than one cached value is returned, the requester chooses the freshest copy.

Writes are not done this way, because it may be better to lose a node and its state than to do a potentially inconsistent proxy write. Most applications that can benefit from OSR, like vehicle tracking, seem to benefit from this access policy: updating nodes proactively broadcast their updated state, while readers contend for opportunistic access. The variables for which updates are broadcast are determined from the programmer's annotation, according to the format specified in Section 4.1. If, after two timeouts, no answer is returned, OSR returns a hard error to the application. Then, Kairos can either invoke CRR, or deliver a fatal exception to the application. Thus, while useful, the chief limitation of this mechanism is that it is restricted only to providing recovery from Transient Failures.

3.2.2 Checkpoint Rollback Recovery Mechanism

CRR can be used to recover a macroprogram across a broad class of failures by *checkpointing* the complete application state at those code points in the macroprogram that enclose program actions involving a vulnerable set of nodes. If the node set changes anywhere within a given scope, the macroprogram can rollback to the most recent saved state and resume the computation. This effectively causes the computation to go back in time, thereby *losing* global work done between the checkpointed time and the time when CRR is invoked. For example, in Figure 2, taking a checkpoint before line 13, and invoking CRR anywhere between lines 13–20 would ensure that the program correctly recovers from node software, node hardware, and network partition failures. By "correct", we mean that the distributed global program state is consistent, and that degraded output guarantees can be maintained.

CRR instances can be used anywhere in the program. They can be nested, or used linearly to improve performance. The exact syntax a programmer uses for this purpose is described in Section 4.1. The only difference between OSR and CRR with regard to programmer-visible usage is that, unlike OSR, CRR does not take any arguments on invocation.

In our implementation of CRR, checkpoints form a hierarchically organized tree if multiple checkpoints are declared within CRR. However, only the checkpoints along the spline from the latest checkpoint in the current scope to the root are pertinent. CRR is thus dynamically scoped, and the semantics of checkpoint activation and lifetime are exactly those of standard stack-based function calling. Hence, syntactically, CRR does not need any additional arguments—it always rolls back to the last relevant frame within the currently active hierarchy. Non-relevant checkpoints may be garbage collected. This behavior is illustrated in Figure 3, where checkpoints are assumed to be taken just before the `create_list()`, `g()`, and `h()` calls. This means that there are at most two active checkpoint frames at any point within `main()`, and the checkpoint management process is very similar to the corresponding function calling.

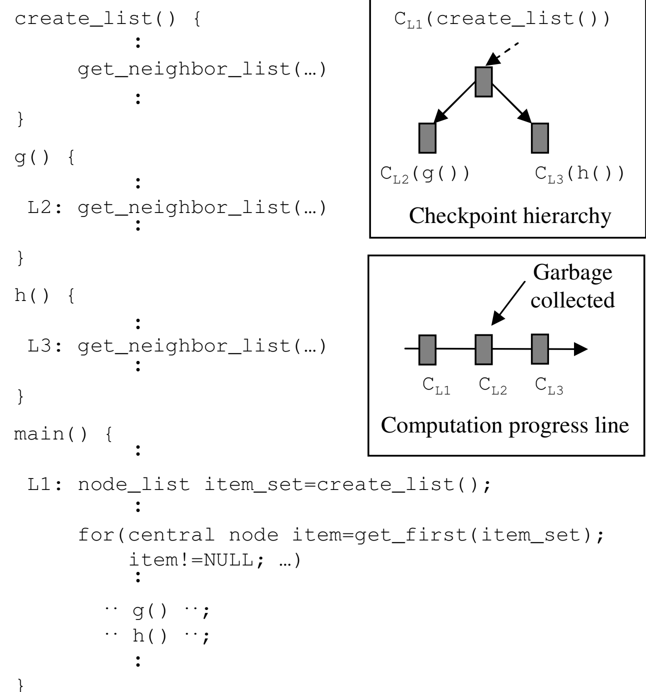

Figure 3: Illustration of Hierarchical Checkpointing in Nested Function Calls, and Their Unraveled Linear Execution Version

When the `Checkpoint` component of CRR is invoked at some place it the macroprogram, it causes *local* checkpoints to be taken at every sensor node, and only one network-wide synchronization occurs. Likewise, when the `Rollback` component of CRR is invoked, nodes locally restore their checkpoints, costing another network-wide synchronization. They are expensive on memory (exact overhead depends on

the nesting depth), which we quantify in Section 5. This is because checkpoints are for the entire in-memory process image at a node, and not just the node's local and central variable data. We leave it to future work to consider how to economize memory by using ideas from the message-based recovery literature of Recoverability Theory [6]: our goal here is primarily to examine whether the CRR mechanism and a strategy like transparent recovery can be made to work for some common sensor network applications.

3.2.3 Partition Recovery Mechanism

The main problem with CRR is its "lose computation work" syndrome, in which the computation progress line is set back in time, and work is lost by the macroprogram in the time between the last checkpoint and when it invoked recovery. Also, in sensor networks, network partitions may occur frequently. If faults affect single nodes, invoking CRR is the right choice if we want to ensure consistency of the macroprogram state. In the case of many continuous-output applications, such as vehicle tracking or the data routing tree in Figure 2, it also happens to be the optimal choice because it always causes the macroprogram to respond both rapidly and correctly to network dynamics, and compute the continuous output with no loss of accuracy.

Unfortunately, for long-running macroprograms that accumulate a lot of program state over time, this "lose work" paradigm of CRR is almost always sub-optimal. For example, consider the simple macroprogram in Figure 4, which computes the average temperature of a region over the course of many days. The function `compute_longterm_av()` is a direct application of Kairos' constructs and idioms.

```
 1: void compute_longterm_av() {
 2:  central node_list full_node_set;
 3:  central node iter;
 4:  central uint sleep_interval=1000, temp=0;
 5   central uint sample_cnt=0;
 6:  full_node_set=get_available_nodes();
 7:  for (;;) {
 8:     sleep(sleep_interval);
 9:     for (node iter=get_first(full_node_set);
              iter!=NULL;
              iter=get_next(full_node_set))
10:     temp=(temp*sample_cnt+t@iter)/
                ++sample_cnt;
11:  }

12: merge void merge_temp(central uint temp_P1,
13:       central uint temp_old_P1,
          central uint sample_cnt_P1,
          central uint sample_cnt_old_P2)   {
14:   temp = (temp_P1*sample_cnt_P1+
               temp_P2*sample_cnt_P2)/
            (sample_cnt_P1+sample_cnt_P2);
      sample_cnt = sample_cnt_P1+sample_cnt_P2;
15: }
```

Figure 4: Example Macroprogram for Computing Longterm Averages

Suppose a network partition occurs between lines 7–11. Kairos would rollback the computation to before line 6, and restart it independently on both halves of the partition. 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 a mechanism called *Partition Recovery* that uses CRR along with a *merge function* that combines the global work done by each group during the partition. The goal is to preserve the values of the long-lived program state variables. In Figure 4, such state is `temp` and `sample_cnt`. Syntactically, a merge function is defined by prefixing the attribute keyword `merge` to a function definition. Semantically, it is called by CRR during recovery, after passing the right parameters. It can access any variables that are accessible at the point to which recovery would shortly undo the computation (*e.g.,* to just before line 7 in Figure 4). Such variables in each of the two macroprograms on the two partition groups can be named within the merge functions by suffixing the macroprogram variable names with `_P1` and `_P2` respectively. It can be seen that this technique is general and flexible, and does not impose any serious restrictions on what type of merge procedures can be used. Figure 5 shows some scenarios that can be reused in a variety of applications.

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 5: Sample Tasks and Partition Recovery Functions

3.3 Failure Classification Heuristic

It turns out that these automated mechanisms are general and sufficient enough to be adaptable to a wide class of failures: we can evolve a one-to-one correspondence between a failure type and a recovery mechanism. This means that it helps the programmer to have Kairos provide a failure identification and classification heuristic, and a suite of automated recovery mechanisms that Kairos can then automatically pair with a failure class. Then, at runtime, Kairos detects a fault, classifies it into a failure type, and invokes the matching recovery method. The programmer's job is thus restricted to specifying which code fragments should be protected against which kind of failures.

There are two main challenges in any failure classification heuristic: a) universality (the ability to detect all classes of faults), and b) low misclassification probability. Regarding

a), Kairos tries to detect all major classes of faults that can typically occur in WSNs. Regarding b), Kairos organizes failures into reliably detectable and non-overlapping failure classes. Together, these two requirements mean we do not deal with malicious byzantine failures [17]. Finally, this detection and classification heuristic achieves low overhead and low latency by commingling its actions with the normal macroprogram execution. Our classifier can deal with the following faults:

Transient Faults Transient faults occur when a read or write to a remote node fails with three successive timeouts, but when the failed node is still in the routing table entries of nodes in the network. In the current implementation (Section 5.1), Kairos uses the services of an underlying routing layer, and reads and writes are implemented as individual messages with end-to-end acknowledgment per message. The timeout period is set to 3 seconds, which is a a reasonable upper-bound on realworld latencies given the average number of hops in today's multihop WSNs (<6), and the average end-to-end per-node goodput under lossy, congested conditions (~ 0.5 packets per second). In TinyOS, unlike in the Internet, the default multihop routing layer constantly observes packet transmissions to determine new routes on demand, which can be a problem for classifying "packet send" failures into transient faults or one of the other kind of faults below. Currently, we avoid this issue by using manual routing tables that timeout and refresh every two minutes, and we leave it to future work to address this issue better. The message overhead for detection and classification of Transient Faults is minimal, and the detection protocol is piggybacked on normal data flow.

Kairos classifies faults of this type as transient failures, and assigns OSR as the best-matched automated recovery mechanism. (Section 3.2.1).

Software Faults This fault arises when a compiled application at a node can encounter various problems, and stops participating in the macroprogram execution when the node itself is alive. We assume, however, that the Kairos node runtime itself is a reliable software component. This means that the system can unambiguously detect this condition when the Kairos runtime at a node responds to status request control messages, but when the application at that node does not take part in control flow synchronization. (For example, periodically, nodes synchronize at line 14 in Figure 2.) Note that remote data access requests may still be successfully fulfilled by the node's Kairos runtime during this period, so we cannot use this method to detect faults in this class. Thus, this protocol is also not message intensive (one message exchange during control flow synchronization), and naturally inlined with the normal execution to have low latencies in identification and classification. Thus, this class of failures is an example of the fail-stop category of failures in general distributed systems [1, 3], in which a node's computation halts, but that such a condition can be reliably detected.

Kairos classifies faults of this type as software failures, and assigns CRR as the best-matched automated recovery mechanism.

Node Hardware Faults The detection technique here is similar to the previous case, except that the node's runtime does not respond to the "send status request" control messages from the macroprogram after a failure from the node's application to participate in the application synchronization. We use timeouts similar to the Transient Faults case. We note that there is a low chance of misclassification here: if the fault is really a Node Software Fault, but if the Kairos runtime is experiencing transient failures, the system may instead falsely diagnose it as a Hardware Fault. This is the only instance of possible misclassification that cannot be reliably detected (interestingly, any routing protocol must also deal with a similar problem). Thus, this class is an example of the crash-stop model of failures [1, 3]. If the CRR recovery mechanism is employed, this node is dropped from the macroprogram. However, the consequences of this misclassification are not severe in terms of application fidelity, as routing would inform the macroprogram through an upcall that a new node is available, which means we would rollback once again to include the node, and the only major cost is two wasteful rollbacks.

Kairos classifies faults of this type as hardware failures, and assigns CRR as the best-matched automated recovery mechanism.

Network Partitions The detection and classification technique is largely similar to the Hardware Faults case above, except that multiple nodes on one side of the partition would be simultaneously unavailable to the other side. The cost of detection is one message per missing node. The issue of misclassification is not relevant here because the probability of multiple nodes undergoing simultaneous independent application crashes and experiencing independent transient failures is minuscule, at least in the scenarios we considered in Section 5.

The typical option, after each partition set has identified this condition, is for them to independently rollback and restart computation from the last checkpoint, and merge their work using PR after they rejoin. Also, in order to completely eliminate misclassification, the runtimes in each group can record enough information to determine, after partition merging, whether their classification is correct, so that the final output is correct, even in the extremely rare case of misclassification. This simple book-keeping also helps to correctly deal with multiple simultaneous partitions.

Kairos classifies faults of this type as partition failures, and assigns PR as the best-matched automated recovery mechanism.

Invariant Violations The last major class of faults known as non-malicious Byzantine failures [1, 3] can be reliably detected and classified by checking programmer written global invariants. This class of faults encompasses logical faults produced by non-malicious software components, such as a buggy macroprogram or non-robust system components.

An example of the latter is unreliable network stacks in WSNs, which, combined with unreliable links with low-strength CRCs, let malformed packets slip through to the application [18]. Common programmer bugs like "off-by-one" are another example. A macroprogrammer can specify invariants as an application-level safeguard, using predicate assertion annotations. An example is the requirement that the variable `dist_from_root@n` in Figure 2 be either `<=max_network_diameter` or $= \infty$.

As Kairos executes the macroprogram, such an assertion is checked for violations, and it can invoke CRR, which is sufficient in many cases [10]. The message cost is a function of the execution frequency and complexity of the logical predicate. Ideally, we would like to take a checkpoint, using CRR, just before the invariant. However, since an invariant can occur within a hotspot region of the code (such as tight loops), we use a simple heuristic in which the Kairos compiler sets the checkpoint to be the code point just before the innermost *for()* within which the invariant appears. We deal with the program analysis issues involved in such tasks in Section 4.2.

Kairos classifies faults of this type as partition failures, and assigns IR as the best-matched automated recovery mechanism.

4. AUTOMATED RECOVERY STRATEGIES

In this section, we consider two questions to drive the usage of the automated mechanisms described in Section 3.2: how to make the mechanisms programmer-friendly, and to what extent and how can they be made transparent.

4.1 Directed Recovery Strategy

We propose a novel strategy called *Directed Recovery (DR)* for answering the first question. DR makes the five recovery mechanisms more accessible to the programmer by letting her *declaratively* invoke them with the appropriate arguments at certain well-defined places in the macroprogram.

DR is attractive because of the code simplicity and economy it can achieve. Another advantage is that the programmer can *structure* recovery usage so that she can easily tailor the recovery mechanisms according to her needs, by nesting the declarations naturally within each other, and closely mirroring the structure of the core program logic: for example, in vehicle tracking [16], she can use the "retry"-based OSR automated mechanism to protect against transient failures at the neighbors of the current master tracker, inside a `for(){}` loop covered by a "repair"-based CRR automated mechanism to protect against the loss of the master itself.

DR uses annotations within the macroprogram to specify the recover mechanisms to use. Such an annotation can appear anywhere in the program, but typically precedes a `for(){}` loop iterator on a `node_list`. For example, suppose the programmer of the code in Figure 2 desires recovery from all transient failures on a node's neighbors variable `iter2` within the `for(){}` block between lines 17–20, and also recovery from all failures on variable `iter1` between 16–20. She could annotate this requirement as $\langle$`TF, iter2`$\rangle\}$ before line 17, and nesting it within $\langle$`*, iter1`$\rangle\}$ before line 13.

Syntactically, the first item in the annotation is an enumerated unordered tuple of failures classes, but can also be the wildcard character `*`. As we saw in Section 3.3, Kairos can accurately identify the correct failure type and assign a recovery mechanism. Kairos can make simple compile-time checks to make sure that the programmer has specified valid recovery arguments in the second item of the annotation.

The semantics are that, once Kairos identifies and classifies a failure at a certain statement in the macroprogram, DR checks the failure against a pertinent annotation's list of failures to respond to. If an exact match by failure name is found in the annotation, DR invokes the corresponding recovery mechanism according to the Failure Type $\Rightarrow$ Recovery Mechanism mapping rules described in Section 3.3. DR also passes the arguments supplied in the annotation to the recovery mechanism at this time.

4.2 Transparent Recovery Strategy

A programmer has to be aware of the macroprogram semantics when using Directed Recovery, because a wrong outcome is possible otherwise. For example, if the programmer of Figure 2 put the annotation $\langle$`*, iter1`$\rangle\}$ before line 16 instead of line 13, this would cause checkpoints to be repeatedly taken during runtime, and these checkpoints would not help to recover from a failure. For this reason, we also explore a *Transparent Recovery (TR)* strategy, in which the system automatically determines the proper place from which to invoke an appropriate recovery mechanism. This task involves identifying all places in the macroprogram that are likely to be affected by recovery, and an algorithm to pick a safe code point to rollback to that would result in the least amount of lost work.

The first task of identifying all and only those code points that might be impacted is made possible by Kairos separating the program state into central variables and node-local variables. We can tell from a simple static analysis of the program where the node-local variables are written to. We can conservatively take a checkpoint during runtime each time we hit such a place during runtime. More sophisticated forms of program analysis can render obsolete many instances of these checkpoints, and we leave it to future work to consider them.

The second task of selecting the correct code point to rollback to is achieved by Kairos as follows: the runtime attaches to each (node-local and central) variable a list of nodes `n` whose node-local data was used to update the variable since the last recovery epoch. This list gets augmented each time the variable is updated using data from a node not currently on the list.

Recall from Figure 3 that the currently active hierarchy of

checkpoints is arranged as a linear list. Therefore, when TR initiates recovery for a failed node, it looks back in the sequence of checkpoint frames for the latest checkpoint without any contribution from `n`. We notice that this search can be carried out at each node in a decentralized manner. Such a checkpoint always exists, with the start of execution serving as a default. It is easy to see that, assuming checkpoints are taken at all vulnerable statically scoped code points before the current code point, this technique finds the checkpoint that would cause the least amount of computational work to be discarded. We leave it to future work to find optimizations that would cause fewer checkpoints to be taken by a more careful program analysis, and that would optimize the data that needs to be kept within each checkpoint for each program variable.

5. EVALUATION

Our evaluation is driven by four key questions about our proposed recovery mechanisms and strategies:

1. How does the Transparent Recovery (TR) strategy compare against simple Manual Recovery (MR) strategies in a workload consisting of the failure types described in Section 3.3? We consider this question from the perspective of application performance as defined by the *accuracy* and *availability* of results delivered by a recovered application. The application user then has two "ground truth" situations two validate these results: a base-case Kairos application encountering no faults, and the same application in the presence of faults.

Our key finding is that transparent recovery can ensure worst-case correctness of the recovered application to within 40% of that of the non-faulting application across a wide range and severity of software and hardware faults, and can improve an application's availability in the presence of faults by a factor of up to 2.75. On the other hand, for certain classes of failures, MR can result in arbitrarily large error and no improvement in application availability compared to a faulting unrecovered application. These results are against a representative set of three sensornet applications.

2. How does the Directed Recovery (DR) strategy compare against TR and MR in the same scenarios used to answer question 1 above? In particular, does it achieve its goal of allowing succinct expression of recovery mechanisms?

Our finding here is that DR achieves expressivity by requiring at most 7 declarations from the programmer for each of the three examples. TR, as expected, does not require any input from the programmer, while even simple MR mechanisms require twice the amount of procedural recovery code.

We also find that DR performs similar to TR with respect to accuracy and availability, but marginally better than TR on resource metrics like message cost and recovery latency because of the small algorithmic runtime overhead of TR, as described in Section 4.2. So, we do not show the performance of DR separately from TR in Section 5.2.

3. How well does the failure classification heuristic perform? Understanding its behavior and performance is crucial because it directly affects the recovery mechanisms that are driven by TR and DR. It is therefore important to ensure that it does not pass incorrect failure type information to TR or DR.

We find that it correctly identifies all faulting situations, and classifies them into failures with misclassification rates less than 5.1% in all cases.

4. What are the performance characteristics of the recovery mechanisms OSR, CRR, PR, and MR? Since TR and DR drive these mechanisms, it is important to understand what their performance is under different failures and application characteristics. For performance metrics, we consider network message cost, recovery latency, and memory overhead.

We find that, as expected, OSR has the lowest recovery latencies among all four generic mechanisms ($\sim 20\%$ of MR, 10% of CRR, and 5% of PR), but is obviously restricted to transient failures. It also has the the lowest messaging cost among all four ($\sim \frac{1}{3}$ of MR and CRR, and $\frac{1}{8}$ of PR). Also, PR's messaging performance varies widely (by a factor of $3\times$) depending on the nature of the application. Finally, while MR requires very little excess memory for recovery, OSR is also lean ($< 4\%$), while CRR and PR require $\sim 2.5\times$ more local memory at a node.

These results are across three representative sensornet applications written in Kairos, the pseudocode for two of which is given in [11]. The three applications are: vehicle tracking for which an explicitly distributed algorithm based on Bayesian belief propagation is given in Liu *et al.* [16]; node localization in a sensor network for which, again, a distributed algorithm based on a cooperative multi-lateration strategy is given in Savvides *et al.* [24]; and quantile estimation for which an explicitly distributed algorithm based on the concept of a summarizing data structure called q-digests is given in Shrivastava *et al.* [25].

Vehicle tracking is an instance of a locally-communicating, continuous-sensing, latency sensitive, network 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 an instance of a network-wide locally-communicating application type, whose output and latency sensitivity requirements can be customized: for bulk data gathering, it can be configured to be a latency tolerant, continuous output application, or for low frequency rare event event monitoring, it can be configured as a latency sensitive single-shot application. Thus, these applications place wide ranging demands on Kairos.

5.1 Implementation, Test Bed Setup, and Evaluation Methodology

We implemented Kairos, and the recovery techniques for Kairos partly in Python, using its embedding and extending APIs, and partly in C. We presented the details of our

Kairos implementation in [11]. Kairos runs on a hybrid Stargate [15] and PC platform. While Kairos itself executes on the relatively resource-rich Stargates, we use 15 Mica-Z motes hosted on the Stargates, and 21 Mica-Z motes hanging off a ceiling array to serve as the network interfaces, as shown in Figure 6. This setup allows us to simulate real-world multihop configurations without being severely constrained by the limited memory resources of the current generation of motes.

Figure 6: Bridged Hybrid Mica-Z on a Stargate (center) and Mica-Z on a PC (flanks) Test Bed

Kairos uses EmStar [7] to implement end-to-end reliable routing and topology management. EmStar drives a Mica-Z mote mounted on the Stargate node, as shown in Figure 6. The ceiling array is connected through a multiport serial controller to a standard PC that runs 21 EmStar processes. Each EmStar process controls a single Mica-Z, and is attached to a Kairos process that also runs on the host PC.

In this environment, all nodes are within a single physical hop of each other, but we configure a logically-multihopped 6x6 2D torus topology. Each evaluation run can contain faults or not. Faults are configured by severing some or all of the 4 links randomly at each node. Faulting nodes do not communicate with one another, and the primary network stays connected except during Partition Recovery. An application is started with no faults, and faults are activated immediately after the first call to *get_available_nodes*() has succeeded.

For vehicle tracking, we assumed a constant speed target moving randomly within the 6x6 grid. Other parameters of the algorithm in [16] were scaled to fit the unit square assumption. For localization, coordinates of beacon nodes were randomly perturbed with Gaussian noise according to the parameters in [24]. The q-digest application uses the `buildtree()` example in Figure 2 to set up the routing tree along which the data digests are sent. In order to measure network availability, we additionally configure q-digest to periodically (every 100s) send digests.

For Manual Recovery, we use code to ignore failed reads from neighboring nodes in case of vehicle tracking (*i.e.,* use the "Discard" generic recovery policy). We wrote code to rebuild the routing tree or restart localization in the case of q-digest and localization respectively (*i.e.,* use the "Redo" generic recovery policy). Since we use a single policy per application, MR serves as both a strategy and a mechanism here.

5.2 Results

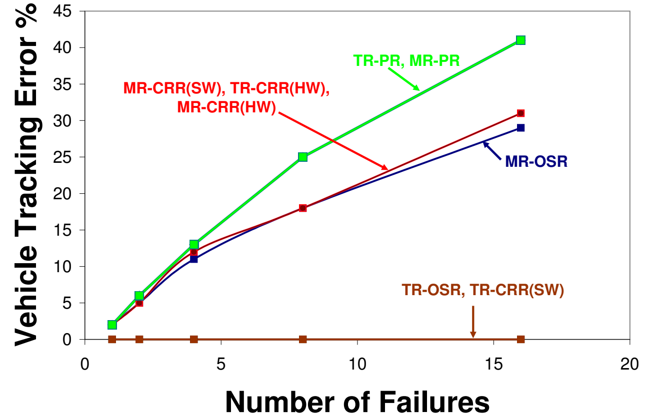

Figure 7: Accuracy comparison of Transparent (TR), Directed (DR), and Manual (MR) Recovery Strategies with Increasing Node Failures

In Figure 7, we plot the relative error in the position estimate computed by TR and MR as compared to the non-faulting tracking application. TR-CRR(SW) gives the performance of TR in the presence of software node faults, which requires a CRR(SW) recovery mechanism. The x-axis is the number of failed nodes. Likewise, CRR(HW) is for pure hardware faults. It is clear that TR can significantly outperform MR in case of OSR and CRR(SW) because tracking does not globally synchronize, and so the Kairos runtime at failed nodes, which is responsible for managing objects including sensor samples, is still alive to supply them to a requesting neighboring master. Both MR and TR perform comparably in the presence of partitions or hardware faults because samples are equally unavailable. Also, while not shown here, we can achieve nearly 100% vehicle tracking error with MR when garbled samples are used, but TR can be protected with invariants.

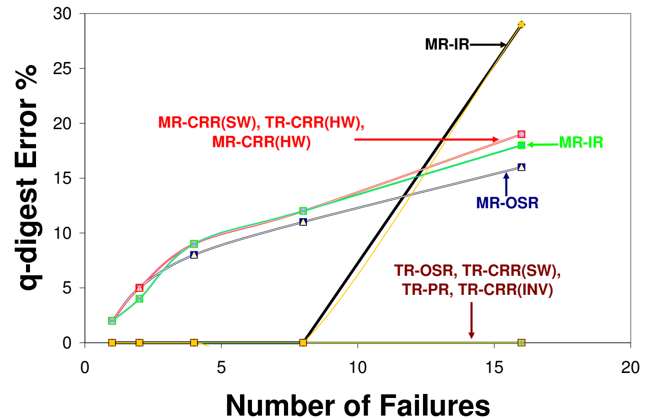

Figure 8: Accuracy comparison of Transparent (TR), Directed (DR), and Manual (MR) Recovery Strategies with Increasing Node Failures

In Figure 8, we plot $\frac{|z_N - z_R|}{|z - z_N|}$, where z is the physical sample median, z_N is the approximation computed by non-faulting q-digest, and z_R is the approximation computed by recovered TR or MR. (This is the same formula used in Figure 7 as well, with the z's denoting estimated and actual vehicle positions.) The interesting difference between this one and Figure 7 is that PR handles partition merges without application errors by merging two q-digests, while it is meaningless to merge a constantly changing output like vehicle position after the partition has healed. Also, since vehicle tracking can involve more inaccurate sampling than q-digest, errors here are smaller and grow more slowly.

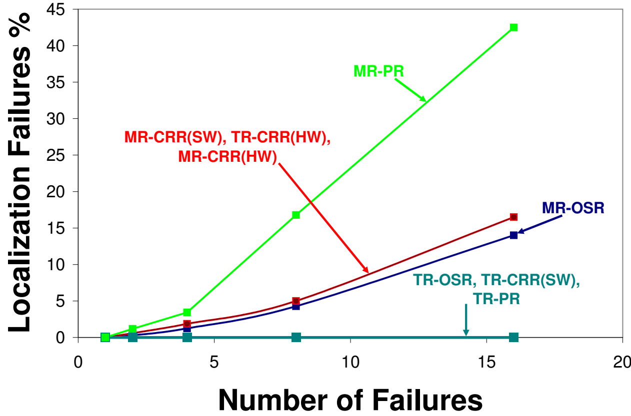

Figure 9: Accuracy Comparison of Transparent (TR), Directed (DR), and Manual (MR) Recovery Strategies with Increasing Node Failures

In Figures 9, we once again see the relative errors for MR and TR. Because localization involves more distant internode communication with increasing node loss, the error curves here are concave and grow faster compared to the previous two. Here, we also need another physical error metric for the fraction of nodes that were unable to localize after recovery compared to the base case, in addition to the higher localizing errors if a node can be localized. This behavior is shown in Figure 10.

In the next two experiments (Figures 11 and 12), we compare the availability factor of results in the presence of faults using the availability metric formula $\log_{10}(\frac{U_F}{U_R})$. Here, U_F, U_R are the fractions of time an unrecovered and recovered application respectively are unavailable. For example, an application with 0.999 availability would $1\times$ (or 100% more) as available as one with 0.99 availability. Clearly, the OSR mechanism and the TR strategy incorporating it can benefit heavily on this metric.

The reason TR-CRR(SW), TR-CRR(INV), TR-CRR(SW) also achieve higher availability over their corresponding MR variants is that, when the master node in the case of vehicle tracking or an internal tree node in the case of q-digest are affected by a fault, the TR-CRR variants can affectively recover from this fatal condition by rolling back to the star, and selecting a new master. The MR variants cannot achieve

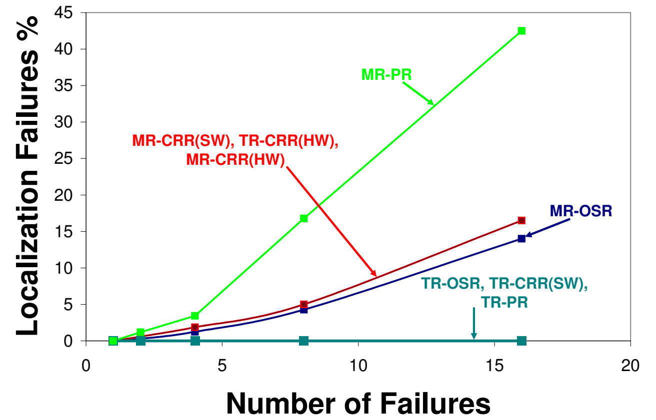

Figure 10: Accuracy Comparison of Transparent (TR), Directed (DR), and Manual (MR) Recovery Strategies with Increasing Node Failures

this, and are thus no more available than the base case with faults but no recovery support.

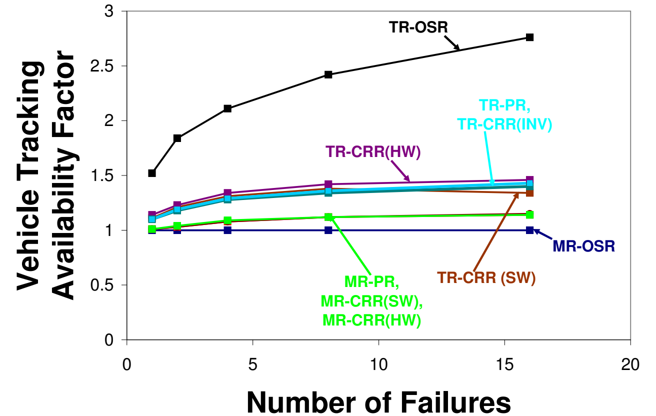

Figure 11: Availability Comparison of Transparent (TR), Directed (DR), and Manual (MR) Recovery Strategies with Increasing Node Failures

In the next experiment (Figures 13), we show the overheads of the OSR, CRR(SW), CRR(HW), CRR(INV), and PR mechanisms. The error bars in the figure are their variation with the number of faults, and are interesting because they indicate that the communication overhead of these mechanisms is independent of the severity of faults, and depends mostly on the nature of the application. As to be expected, OSR has universally low message overhead. CRR(SW) and CRR(HW) are almost equal because they share the same logic when invoked. CRR(INV) involves slightly more work than either. PR is almost twice as heavier than CRR for some applications like q-digest that span the entire network, and, therefore, involve a large number of nodes. For locally communication applications like vehicle tracking, on the other hand, the overhead is almost a constant. We also note that all automated mechanisms except PR are well within 10% of MR, and the total message cost

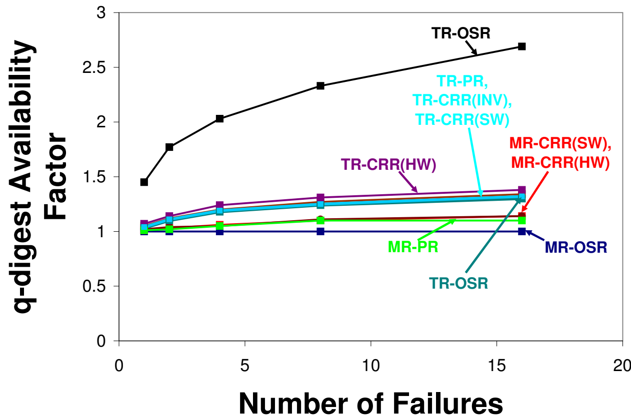

Figure 12: Availability Comparison of Transparent (TR), Directed (DR), and Manual (MR) Recovery Strategies with Increasing Node Failures

of an application with PR is within 40% of that of the total message cost with MR. Since PR delivers clear application benefits over MR, this overhead seems acceptable.

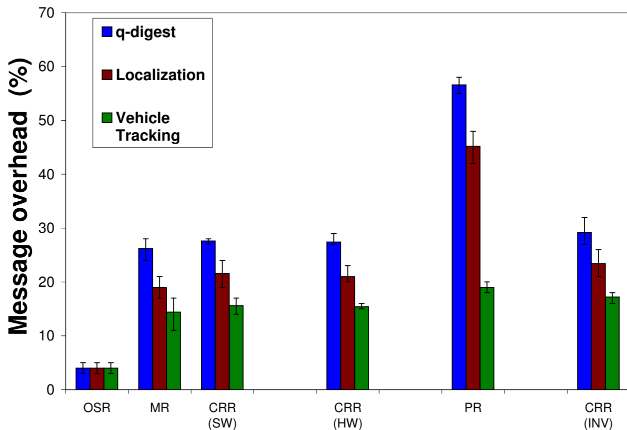

Figure 13: Message Overhead Comparison of OSR, CRR, PR, and MR Mechanisms

In Figure 14, we show that CRR requires $\sim 2.2 - 2.5X$ memory, and is almost independent of the number of faults, as well as the application characteristics. Instead, it only depends on the average nesting depth of checkpoints, which is slightly more than 2 in all cases.

In Figure 15, we see that the misclassification rate is greatest for localization, and depends almost solely on the communication pattern (which is lower for local communication), which is to be expected. The number of failures once again does not make much difference except for localization.

We end this section with the following recommendations for application designers: OSR mechanism works well for applications that are latency sensitive. CRR is good for continuous output applications, while those requiring longevity should consider PR as well. CRR with invariants is also a must for high fidelity applications.

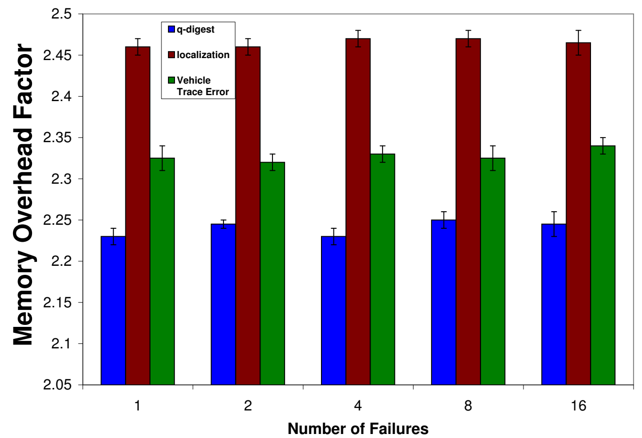

Figure 14: Memory Overheard for CRR

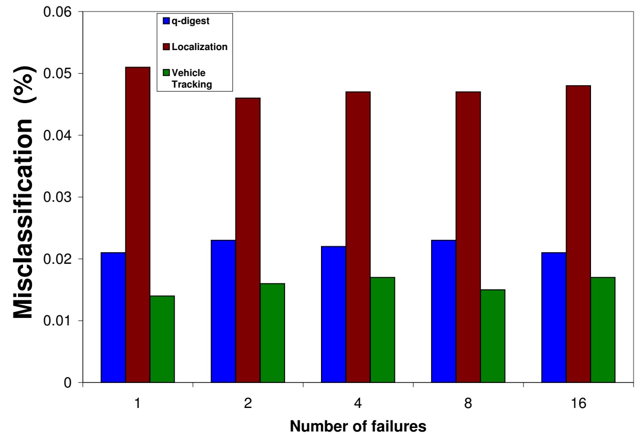

Figure 15: Misclassification Percentage of Failure Classification Heuristic

6. RELATED WORK

Macroprogramming is a relatively recent concept in sensor networks [23, 29, 28], and we are not aware of any prior work that considered programmable failure recovery primitives, techniques and strategies in sensor networks. We believe that key concepts behind mechanisms like OSR and CRR, strategies like TR, and the failure classification heuristic can be used in other macroprogramming models like Regiment [28] that is also vulnerable to faults due to a lack of atomicity guarantees. For example, the common idiom `read node = (read sensor PROXIMITY node, get coords node)` in [28] assumes the `read` and `get` Haskell-style functions used in this `read node` function definition are mutually atomic on a node. We therefore see our work as complementing these other macroprogramming models as well.

Kairos borrows heavily from prior work in parallel and distributed systems. With respect to its programming model, its concept of central variables bears some resemblance to Linda [5]'s tuple space based coordination, and to Split-C [2]'s split local/global shared memory model. The prime distinction between Kairos and all other previously proposed

systems that we are aware of lies in its novel programming model, which consists of of a sequential tasking model on a centralized memory model. Therefore, a macroprogram in Kairos is not explicitly parallelized and does not use synchronization primitives, unlike many parallel and distributed systems. Since Kairos tries to achieve network economy rather than the traditional goals of maximum concurrency, throughput, or response times, its implementation strategies are also different, and include macroprogram transformation plans for in-network placement and execution. Extracting the last ounce of parallelism out of our computation model is an explicit non-goal, so we hope to avoid most of the intractable issues that have stymied general-purpose parallel, distributed, or concurrent programming models aspiring to that prime objective.

Recoverability Theory [6] is relevant to the recovery mechanisms we present in this paper. Many of the fundamental concepts we used extensively in our recovery mechanisms, strategies, and heuristic, like checkpointing, promiscuous caching, program analysis and annotations are all well-studied subjects. The distinguishing feature of our work from this immense prior body of literature is how we exploit the synergy between the language, the compiler, the runtime, and the sensor network domain to design novel recovery primitives that are simple, feasible, and have semantic significance.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have taken the first steps toward macroprogramming facilities for recovery in sensor networks. We proposed two generic automated mechanisms, and described a heuristic to classify faults and invoke these mechanisms. We also explored two promising strategies to drive these mechanisms: a declarative strategy and a transparent recovery strategy. Transparent recovery, in particular, seems promising in the scenarios we evaluated, but we need to do more experimental work with a wider range of applications, and gather more experience about realworld deployments. A number of sub-components described in this paper also need more critical study: examples are the program analysis compiler algorithm and the checkpoint maintenance runtime algorithm used in transparent recovery.

8. REFERENCES

[1] http://www.cs.cornell.edu/Courses/cs514/2005sp/Slide%20Sets.htm.

[2] http://www.cs.berkeley.edu/projects/parallel/castle/split-c/.

[3] Cs514: Intermediate course in operating systems. http://www.cs.cornell.edu/Courses/cs514/2005sp/lec-11.ppt.

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

[5] N. Carriero, D. Gelernter, and T. Mattson. The linda alternative to message-passing systems. *Parallel Computing, 20: 458-633, 1994*.

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

[7] J. Elson, S. Bien, N. Busek, V. Bychkovskiy, A. Cerpa, D. Ganesan, L. Girod, B. Greenstein, T. Schoellhammer, T. Stathopoulos, and D. Estrin. Emstar: An environment for developing wireless embedded systems software. *CENS-TR-9, 2003*.

[8] W. F. Fung, D. Sun, and J. Gehrke. Cougar: the network is the database. *SIGMOD, 2002*.

[9] D. Gay, P. Levis, R. Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. *PLDI, 2003*.

[10] J. Gray. Why do computers stop and what can be done about it? *SRDS, 1986*.

[11] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using kairos. *DCOSS 2005*.

[12] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D.Culler, and K. Pister. System architecture directions for network sensors. *ASPLOS, 2000*.

[13] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. *SenSys, 2004*.

[14] Crossbow Technology Inc. Mica2dot series (mpr5x0). http://www.xbow.com/Products/productsdetails.aspx?sid=73.

[15] Crossbow Technology Inc. Stargate platform. http://www.xbow.com/Products/XScale.htm.

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

[17] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem.

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

[19] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. *NSDI, 2004*.

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

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

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

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

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

[25] N. Shrivastava, C. Buragohain, S. Suri, and D. Agrawal. Medians and beyond: New aggregation techniques for sensor networks. *SenSys, 2004*.

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

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

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

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