# Resilient Sensor Network Query Processing Using Logical Overlays

Alan B. Stokes
School of Computer Science
University of Manchester
Manchester M13 9PL, UK
stokesa6@cs.man.ac.uk

Alvaro A.A Fernandes
School of Computer Science
University of Manchester
Manchester M13 9PL, UK
alvaro@cs.man.ac.uk

Norman W. Paton
School of Computer Science
University of Manchester
Manchester M13 9PL, UK
norm@cs.man.ac.uk

## ABSTRACT

The typical nodes used in mote-level wireless sensor networks (WSNs) are often brittle and severely resource-constrained. In particular, nodes are often battery-powered, thereby making energy depletion a significant risk. When changes to the connectivity graph occur as a result of node failure, the overall computation may collapse unless it is capable of adapting to the new WSN state. Sensor network query processors (SNQPs) construe a WSN as a distributed, continuous query platform where the streams of sensed values constitute the logical extents of interest. Crucially, in the context of this paper, they must make assumptions about the connectivity graph of the WSN at compile time that are likely not to hold for the lifetime of the compiled query evaluation plan (QEP) the SNQPs generate. This paper addresses the problem of extending the lifetime of an evaluating QEP in the event of node failures. The basic idea is to derive an equivalence class over the nodes in the WSN that are equipotent for a given QEP and then to assign each QEP fragment instance to a set of equipotent nodes (rather than a single one). In this respect, the scheduling of QEP fragment instances is onto an overlay network of logical nodes, each of which maps to many physical nodes in the connectivity graph. We contribute a description of how this approach has been implemented in an existing SNQP and present experimental results indicating that it significantly increases the overall lifetime of a query whilst incurring small runtime adaptation costs.

## Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management, Distributed Databases, Query Processing—*Wireless Sensor Networks, Resilience*

## Keywords

Sensor Network Query Processors, Wireless Sensor Networks, Resilience

## 1. INTRODUCTION

Wireless sensor networks (WSNs) are useful in data collection, event detection or entity tracking applications. In particular, mote-level WSNs are sufficiently inexpensive that one can envisage deploying them to sense at fine granularities both over space and over time. With the low cost, however, come severe resource constraints in terms of energy stock, communication range, computational and storage capabilities, etc. Our focus here is on WSNs comprising static motes of this kind (e.g., [2]).

If one views the WSN as simply an instrument for data collection, one might task the relevant subset of nodes to sense the physical world and send the sensed values, using multi-hop communication paths, towards a base station where all the processing takes place. However, sending all data in raw form to the base station causes more bytes to be transmitted than would be the case if the nodes along the route to the base station were tasked with some of the processing [11]. Since the energy cost of processing data is one order of magnitude smaller than the energy cost of transmitting the same data [7], it is more energy-efficient to do as much processing as possible inside the WSN, as this is likely to reduce the number of bytes that are transmitted to the base station.

One approach to in-WSN processing construes the WSN as a distributed database, and the processing task injected into nodes for execution is the evaluation of a query evaluation plan (QEP). In this approach, users specify their data requirements in the form of declarative queries, which the system, called a sensor network query processor (SNQP), compiles into optimized QEPs for injection into the WSN. Through periodic evaluation, a stream of results is returned to the users via the base station.

Many SNQPs have been proposed in the literature, e.g. SNEE [4], TinyDB [10], and AnduIN [8]. These SNQPs often differ in terms, among others, of how much of the required query functionality can be injected into the WSN, how much use they make of distributed query processing techniques (e.g., fragment partitioning, buffering tuples for block transmission, etc.), and how much compile-time knowledge of the WSN state they require in order to produce a QEP. Thus, AnduIN does not inject joins for in-network execution, only QEP leaves, i.e., sensing tasks. AnduIN uses a TCP/IP protocol stack and therefore has no need to know the state of the WSN connectivity graph at compile time. In contrast, TinyDB is capable of performing limited forms of joins inside the WSN and pushes the entire QEP to every participating node. TinyDB infers the current connectivity graph from the dissemination of the QEP into the WSN. Finally, SNEE, which we focus on in this paper, pushes very expressive QEPs into the WSN whilst still partitioning the latter into fragments that are as small as possible for each node. However, SNEE neither uses a generic protocol stack nor can it compile the QEP without knowledge of the current connectivity graph.

SNEE does more in-WSN processing than the other SNQPs mentioned above. It generates QEPs that deliver good energy efficiency [4] whilst scheduling for node execution QEP fragment instances that use less memory (partly by not using, and hence not loading, generic protocol stacks) [4] than the other SNQPs mentioned above. To generate QEPs where medium access, routing, and transport are query-specific, the SNEE compiler takes as input (among other metadata) the current connectivity graph. This implies a further, and stronger, assumption, viz., that if the connectivity graph changes (e.g., a node fails) during the lifetime of QEP $p$, then $p$ may not be optimal for the new topology (and, indeed, $p$ may even be unable to run). In other words, maximizing QEP lifetime is dependent on resilience to failure. A SNEE QEP often has its lifetime bounded by the time-to-failure of participating nodes. In practice, not only node failure is assumed to be a common occurrence, the energy stock of a participating motes is guaranteed to diminish over time and depletion eventually causes the motes to become non-functional.

SNEE QEPs are therefore particularly brittle: if a participating node fails, poor performance, or even a crash, could ensue. One aspect of poor performance is the lack of adaptation to tuple loss when the corresponding extent draws from a failed node. Such failures lead to partial results for the query. It is, therefore, desirable that, if possible, the QEP is adapted in response to node failure. Another possibility is that the failed node causes the communication graph used by the QEP to partition in such a way that, although all sensed values are flowing out of the leaves, they cannot be used as they fail to reach some downstream operators, i.e., the energy expenditure of the acquisitions would be wasted.

In this paper, we break down the process of adapting to node failure into two stages: firstly, we compute new paths for routing tuples around the failed nodes; and secondly, we reschedule the QEP fragment instances that were running on the failed node to nodes in the newly computed paths. Adaptations aim to minimize information loss and foster compliance with quality of service (QoS) expectations such as maximum delivery rate and constant acquisition rate.

The purpose of adaptations, in the case of this paper, is to maximize the lifetime of the QEP. Since lifetime is influenced by the rate of depletion of energy stocks and since any adaptation will cause some such depletion (i.e., carries an energy overhead cost), adaptations must take into account the time taken to adapt (during which, data will cease to flow) and the energy spent in carrying out the adaptation. Our hypothesis is the benefit of adapting with a view to increasing the QEP lifetime (and, therefore, the amount of data produced) outweighs the cost incurred in adapting. We propose an adaptation strategy that takes advantage of (fortuitous or planned) functional redundancy to increase resilience to failure. In particular, at compile time, we compute an overlay network over the physical network, in which one logical node in the former maps to $k + 1$ equipotent nodes in the latter, where $k$ is the desired resilience level. The basic idea is to derive an equivalence class over the nodes in the WSN that are equipotent for a given QEP and then to assign each QEP fragment instance to a set of equipotent nodes (rather than a single one). In this way, for the quality of the QEP to be compromised due to the failure of a node $n$ running a fragment instance $f$, all the $k$ nodes that are equipotent to $n$ must also have failed before $f$ fails. We have studied the above hypothesis experimentally. The results show that the adaptation costs incurred by the overlay strategy do not preclude significant increases in the lifetime of a QEP.

The rest of the paper is as follows: Sec. 2 briefly describes related work. Sec. 3 describes the SNEE SNQP and how the time and energy costs of a QEP are modelled. Sec. 4 describes the logical overlay strategy. Sec. 5 describes how we experimentally evaluated it. Sec. 6 draws conclusions.

## 2. RELATED WORK

Broadly speaking, current SNQPs are not very tolerant of node failure. In TinyDB, the fact that routing trees [10] are constructed during the QEP dissemination process provides some amount of inter-query fault tolerance, as failed nodes do not take part in disseminating the next QEP (which could be a recompilation of the same query) and hence will be disqualified from participating in its evaluation. Also, each node in TinyDB evaluates the entire QEP (i.e., TinyDB makes no attempt to partition the plan into fragments), and, as a result, ignoring a failed node is a sound strategy. Thus, whilst TinyDB is not, strictly speaking, adaptive, it is, to a certain degree, resilient to some forms of node failure. However, tuple transmission is not globally scheduled (as it is in SNEE), so there is no way of estimating how many tuples might be lost as a result of failed nodes.

The SmartCIS project [9] builds upon TinyDB with a particular goal (among others) of supporting fault-tolerant routing trees via multi-path transmissions. This approach incurs energy overheads in verifying that current paths are correct and in searching for new correct ones.

AnduIN has no specific mechanism for fault tolerance. In contrast with both TinyDB and SNEE, which compile into TinyOS [6], AnduIN compiles into Contiki [3]. The difference is relevant in our context because, unlike TinyOS, Contiki provides a TCP/IP-based communication protocol stack. Thus, AnduIN benefits from the robust routing and transport properties built into TCP/IP. The drawback is that TCP/IP incur much greater overheads (and take up more memory footprint) than the minimalistic, query-specific protocols used by TinyDB and SNEE. Some of these overheads stem from the need to maintain up-to-date connectivity paths as well as from the need to send acknowledgement packets. As to memory occupancy, TCP/IP implementations will take up space and will also claim more memory for such structures as routing tables. By reducing the memory on the nodes that can be allocated to the QEP, there is a reduction in how much processing can be shipped to the WSN and how much memory can be used buffering and blocked transmission, both features that are energy-saving. AnduIN does not adapt to failure of acquisition nodes.

The current, publicly-released version of SNEE has no adaptive behaviour. As pointed out, the compilation and optimization of a query takes as input the connectivity graph and of the nodes from which logical extents for sensed streams stem. As a result, any node failure compromises the quality of the compiled QEP: if acquisition (i.e., leaf) nodes fail, data is lost; if non-leaf nodes in the routing tree fail, the QEP execution fails.

## 3. TECHNICAL CONTEXT

SNEE aims to generate energy-efficient QEPs. The compilation/optimization process takes as input a SNEEql query (as exemplified in Fig. 1), QoS expectations (not shown in the figure) in the form of a desired acquisition rate (i.e., the frequency at which sensing takes place) and a maximum delivery time (i.e., an upper bound on the acceptable amount of time between data being acquired and being reflected in the emitted results), and the following kinds of metadata: (1) the current connectivity graph, which describes the (cost-assigned) communication edges in the WSN; (2) the logical schema for the query, which describes the available logical extents over the sensing modalities in the WSN; (3) the physical schema for the query, which describes which physical nodes contribute data to which logical extent, and which node acts as base station; (4) statistics about nodes (e.g., available memory and energy stocks); (5) cost-model parameters (e.g., unit costs for sleeping, sensing, processing, and communicating) [1]. The query takes two streams, one stemming from sensors in a field, the other from sensors in a forest. It joins them on the condition that light levels are lower in the field than in the forest and emits onto the output stream the matching values and the ids of the nodes that generated them.

Fig. 2 shows the SNEE (compilation/optimization) stack. As a distributed query optimizer, it uses a two-phase approach. The single-site phase (Steps 1-3 in Fig. 2) comprises the classical steps needed to compile and optimize a query for centralized execution. The outcome is the physical-algebraic form (PAF) for the query, where each operator has been given its execution order and assigned a concrete algorithm. The multi-site phase (Steps 4-7 in Fig. 2) turns the PAF into a distributed algebraic form (DAF) for the query by making decisions that are specific to in-WSN execution. These include deciding on a routing tree $R$, on fragment instance allocation along the routing tree captured as a DAF $D$ and on timing the activities in the nodes (switching from QEP fragment evaluation to communication and so on) in the form of an agenda $A$. A final step converts the triple $\langle R, D, A\rangle$ into a set of per-node nesC/TinyOS source file, which are then compiled into binary form. This is what we refer to as the executable QEP.

In more detail, Step 4 in Fig. 2 generates a routing tree (RT) for the query as an approximation of a Steiner tree, e.g., the one in Fig. 3(a) for our example query. Each vertex is a sensor node; an edge denotes that the two nodes can communicate; the arrow denotes the direction of communication; double-line circles denote the sink or else nodes that do sensing; single-line nodes only do processing or communication or both. Recall that a Steiner tree is a minimum spanning tree (and hence likely to be energy-efficient) that necessarily includes a given set of nodes. In our case, these are the leaves (i.e., the acquisition nodes) and the root (i.e., the base station).

```
Logical Schema:
  field  (id, time, temp, light);
  forest (id, time, temp, light);

Physical Schema:
  field: {N6, N9}; forest: {N7}; sink: {N8}

Q: SELECT RSTREAM c.id, c.light, f.id, f.light
   FROM   field[NOW] c, forest[NOW] f
   WHERE  c.light < f.light
```

**Figure 1: Example Query, Logical/Physical Schemas**

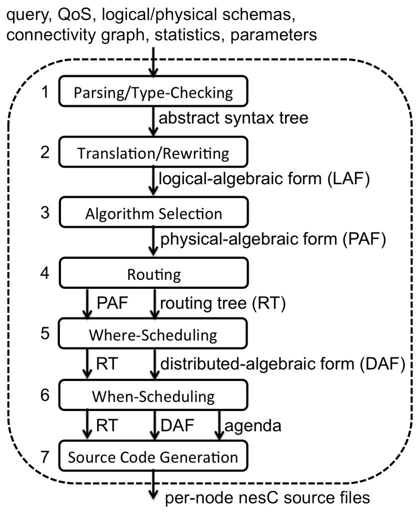
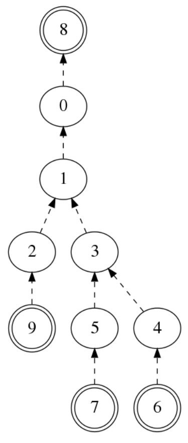

query, QoS, logical/physical schemas, connectivity graph, statistics, parameters
1 Parsing/Type-Checking
abstract syntax tree
2 Translation/Rewriting
logical-algebraic form (LAF)
3 Algorithm Selection
physical-algebraic form (PAF)
4 Routing
PAF
routing tree (RT)
5 Where-Scheduling
RT
distributed-algebraic form (DAF)
6 When-Scheduling
RT
DAF
agenda
7 Source Code Generation
per-node nesC source files

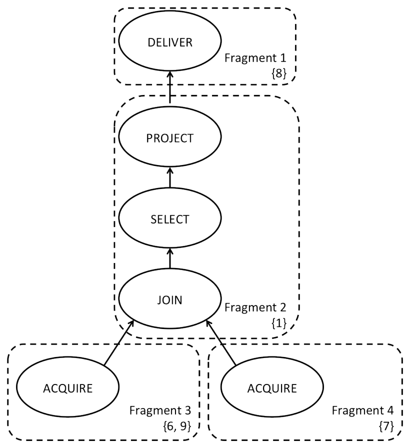
**Figure 2: The SNEE Stack**

Step 5 in Fig. 2 decides which fragment instances to place for execution in which node. This partitions the PAF into fragment instances and assigns the latter to RT nodes with a view to conserving energy by reducing the number of tuples that need to be transmitted. The resulting DAF for the example query is shown in Fig. 3(b). Dashed boxes define fragment boundaries; the list in curly brackets at the bottom-right corner (below the fragment identifier) denotes how many instances of that fragment there are and in which nodes they run. The fragment containing the deliver operator runs on the sink node, the fragment instances containing the acquisition operators run on the leaf nodes and the remaining fragment instances are assigned to run on Node 1 because it is, amongst the nodes through which all tuple streams required for the join flow, the hop-count closest to the leaves. We call such nodes, *confluence nodes*.

Step 6 in Fig. 2 decides when to execute the different tasks in each participating node. These decisions are represented as an agenda, i.e., a matrix where rows denote points in the query evaluation cycle, columns denote participating nodes, and the content of each cell defines the task scheduled for that node at that time. The agenda for the example query is shown in Fig. 3(c). Fragments are identified by number as in Fig. 3(b), with subscripts denoting fragment instances; the notation *txn* (resp., *rxn*) denotes that that node at that time is transmitting to (resp., receiving from) node $n$; a row labelled *sleeping* denote the fact that, for that slot, WSN is idle. In the process of deciding on an agenda, SNEE also determines how much buffering of tuples can be done on the nodes with the memory remaining from fragment instance allocation. SNEE tries to pack tuples into blocks to save on transmission overheads, but since buffering delays delivery,



(a) RT



(b) DAF

| Time | Sites | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| (ms) | 9 | 2 | 7 | 5 | 6 | 4 | 3 | 1 | 0 | 8 |
| 0 | $F3_1$ | | $F4_1$ | | $F3_1$ | | | | | |
| 100 | sleeping | | | | | | | | | |
| 5000 | $F3_2$ | | $F4_2$ | | $F3_2$ | | | | | |
| 5100 | *tx2* | *rx9* | | | | | | | | |
| 5401 | | *tx1* | | | | | | *rx2* | | |
| 5702 | | | *tx5* | *rx7* | | | | | | |
| 6002 | | | | *tx3* | | | *rx5* | | | |
| 6303 | | | | | *tx4* | *rx6* | | | | |
| 6604 | | | | | | *tx3* | *rx4* | | | |
| 6905 | | | | | | | *tx1* | *rx3* | | |
| 7264 | | | | | | | | $F2_1$ | | |
| 7278 | | | | | | | | *tx0* | *rx1* | |
| 7637 | | | | | | | | | *tx8* | *rx0* |
| 7997 | | | | | | | | | | $F1_1$ |
| 8782 | sleeping | | | | | | | | | |

(c) Agenda

**Figure 3: Example Inputs to Code Generation**

it places a limit on the buffering in order to not violate the user-specified maximum delivery time. By being governed by an agenda, a SNEE QEP implements a simple form of TDMA. Whilst this is often economical provided that the estimation models are accurate (and [1] shows that they are), any changes to the timing of the operators or transmissions requires the agenda to be recomputed and hence the QEP to be recompiled and propagated into the WSN.

Step 7 in Fig. 2 takes the RT from Step 4, the DAF from Step 5, and the agenda from Step 6 to produce a set of per-participating-node source files. Compiling these files yields the binaries that, together, comprise the QEP. For an in-depth description of the SNEE compilation stack and data structures, see [4].

We note that, as described in [1], SNEE makes intensive use of the empirically-validated analytical cost models for energy, memory and time expenditure computed over the RT, DAF and agenda for a query. For example, in SNEE, we can estimate the energy and time cost of running a QEP fragment instance on any given node per agenda execution cycle. Such capabilities allow us to estimate the time that a QEP will run before a given node fails due to energy depletion.

## 4. RESILIENCE TO NODE FAILURE

The problem we address can be informally phrased as follows: given a SNEE QEP $q$ (computed as described in the previous section), if a participating node in $q$ fails at run-time, adapt $q$ so that its lifetime is extended. We propose an adaptation strategy that takes advantage of (fortuitous or planned) node redundancy in order to increase $q$'s resilience to node failure. In particular, at compile time, we compute an overlay network over the physical network, in which one logical node in the former maps to $k+1$ equipotent nodes in the latter. The basic idea is to derive an equivalence class over the nodes in the WSN that are functionally equivalent for a given QEP $q$ and then to assign each fragment instance of $q$ to a set of equivalent nodes (rather than a single one). In this way, for $q$ to be compromised due to the failure of a node $n$ running any fragment instance $f$ of $w$, all the $k$ nodes that are equipotent to $n$ must also have failed before $f$ fails and $q$ is compromised.

One desideratum for any solution based on adaptive strategies is that the cost of enacting the adaptive response must be low, so that the net benefit of having adapted is high. The solution contributed in this paper is guided by the desire to minimize run-time reprogramming. To understand why, note that a SNEE QEP is injected into the WSN using an over-the-air-programming (OTAP) layer that routes the appropriate binary to each node and writes them onto the latter's programming memory. We note that both packet transmission and packet writing are costly in both time and energy terms. Since node binaries break down into a large number of transmitted packets, and each such packet must be written onto flash memory, the cost of (re)programming is very high. As such, our aim is to reduce the run-time cost of responding to node failure and, for this purpose, we aim, at compile-time, to preventively replicate code on more nodes than would otherwise be needed and keep the replicas inactive until and unless nodes fail. When they do, the run-time adaptation simply involves activating a (so to speak) dormant node that is equipotent to the failed node.

Thus, our approach to computing the logical overlay aims to make run-time responses reprogramming free: at compile-time we identify equivalence classes of cardinality $k+1$ for each participating node and program the $k+1$ nodes in the same equivalence class with the same binary. At any point in time, the set of $k$ nodes that are equipotent to an active node $n$ has two partitions: the (possibly empty) set of nodes that have failed so far, and the (possibly empty) set of inactive nodes. If, and when, $n$ fails, it is added to the set of failed nodes, and one inactive node is activated to replace it. Thus, the lifetime of the QEP can potentially be extended until all the $k$ inactive replicas have been activated and failed.

This approach leads to very low run-time adaptation cost and, hence, to large net benefit from adaptations. However, it is dependent on there being opportunities for finding equivalences classes in the WSN. This may happen fortuitously in the case of very large scale and dense deployments, or by design, in the case where fault-tolerance is important and therefore controlled redundancy is a requirement on the deployment design. The trade-off, therefore, is between investing in node redundancy and reaping guaranteed longer QEP lifetimes in the presence of node failure or else refraining from investing in redundancy and adapting through reprogramming, which offers less strict assurances of significant extensions in QEP lifetimes.

The remainder of this section describes both the compile-time and run-time changes made to SNEE (as described in [4]) to enable resilient QEP evaluation using logical overlay. The solution takes as input a SNEE QEP $q$ , all the metadata $M$ and all the intermediate data structures $D$ used in compiling $q$. There are four stages in the compile-time part of the process, as follows: given $q$ and $M$, in Stage 1, a super-overlay for $q$ is generated, which, in Stage 2, converted into a set of logical overlays by considering alternative communication edges; then, given $q$, $M$ and $D$, in Stage 3, the logical overlays are ranked using the time, memory and energy cost models; and, finally, in Stage 4, the per-node binaries in $q$ are replicated to the $k$-equipotent nodes of each participating node. The resulting QEP $q'$ differs from the original $q$ in that for each participating node there are $k$ other equipotent nodes in $q$ that are already programmed for performing the functions assigned to it in $q$. The stages above are explained in more detail below with the use of examples.

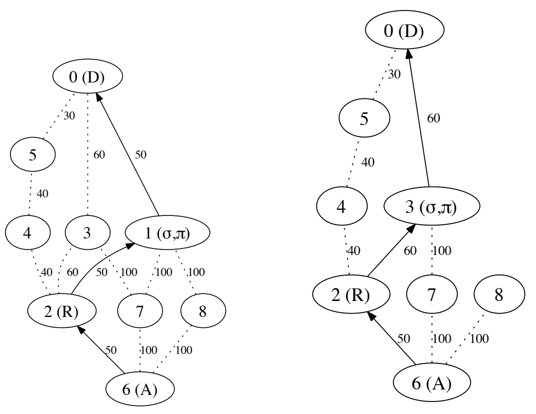
## 4.1 Generating Super-Overlays

We refer to a node in the RT computed by SNEE as an *active node* (in the sense that, by default, it will be running the assigned binary when the initial QEP starts to evaluate). Stage 1 generates super-overlays over the subset $CN \subset AN$ of active nodes in the RT that we refer to as *candidate nodes* for replication. An internal node in $AN$ is a candidate node in $CN$. An acquisition node in $AN$ is a candidate node in $CN$ if it is asserted, in the physical schema, to be equipotent to another node in the WSN. This caters for the situation in which the precise location and sensing electronics of an acquisition node is not sufficiently relevant, and therefore, in case a specific acquisition node fails, others are considered equipotent to it in spite of being in a different location and possibly having different sensing electronics.

A node $w \in W$, where $W$ is the set of all nodes in the WSN that are not already participating in the QEP, is in the same super-overlay node as an active node $c \in CN$ if the following conditions are met: (i) the available memory in $w$ is at least the same as the available memory in $c$ (this is so that $w$ can host the binary that has been assigned to $c$ by SNEE); (ii) if $c$ is an acquisition node, then $c$ and $w$ have been asserted to be equipotent; (iii) if $lp(c)$ and $lc(c)$ denote, resp., the parent and children of $c$ in the super-overlay, then there are edges in the connectivity graph between $w$ and, on the one hand, the active node in $lp(c)$, and, on the other hand, the active nodes in $lc(c)$. In the example in Fig. 4(a), Nodes 1 and 3 satisfy these conditions and therefore form a super-overlay node, as do Nodes 7 and 2, and Nodes 8 and 2. This results in the super-overlay in Fig. 5(a).

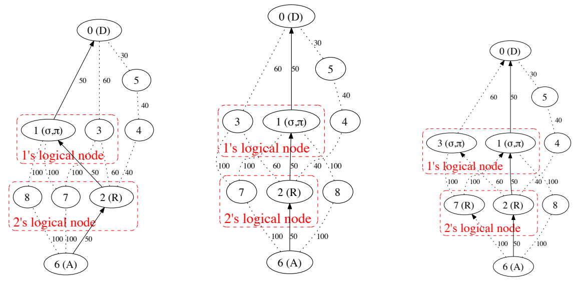
## 4.2 Generating Logical Overlays

When a node $w$ belongs to a super-overlay node $\Omega$, the conditions above ensure that $w$ is connected with the active node (as opposed to every nodes) in the children and the parent super-overlay nodes of $\Omega$. The conditions above do not ensure that $w$ is connected with every node (other than the active one) in the children and the parent super-overlay nodes of $\Omega$. There are several possible permutations in which the physical nodes in one logical node can communicate with the physical nodes in another logical node. The membership of each logical node determines, therefore, whether edges can be formed between logical nodes that are preserved for every pair of physical nodes that can be drawn,



(a) Initial QEP (b) Post-Adaptation QEP

**Figure 4: Initial and Post-Adaptation QEPs**



(a) Super-Overlay (b) Overlay (c) Replicated

**Figure 5: Super-Overlay v. Logical Overlay**

one in turn from each, from the respective logical nodes. In order to determine the best permutation of physical nodes in each logical node, we first identify the complete set of valid permutations and generate, as a result, a set of logical overlays for assessment.

The algorithm starts by traversing the RT from the root until it reaches a node for which a super-overlay node has been formed. All the connecting permutations between the physical nodes in each of the logical nodes are considered with a view to forming logical nodes with size $k + 1$, which denotes the required *resilience level*. Once the parent permutations are generated, the process is repeated to generate the children permutations. In both cases, the correctness condition is that there are $k+1$ physical nodes in every logical node and that if two logical nodes form a logical edge, then each the physical nodes in one logical node forms a physical edge with a physical node in the other logical node.

For every correct permutation that is generated in this way, a candidate logical overlay is generated, and the process proceeds down the RT. If no correct permutation can be generated, the parent permutation is removed from consideration and a new parent permutation is considered. If there are no correct permutations left to assess, the algorithm returns a failure-to-form-overlay result, as a logical node with resilience level $k$ could not be formed for that RT.

In the example, assuming that $k = 1$ and that no equipotent acquisition nodes have been asserted, Node 1 would be located first. Then, the permutation consisting of Node 3 and Node 1 would be generated as it is the only correct permutation, which suffices if $k = 1$. We then look at the logical node whose active node is Node 2 as it is a child of Node 1. We generate the possible permutations with Nodes 7 and 8. As Node 3 can only communicate with Nodes 7 and 2, the only correct permutation is the one formed with these two nodes. As Node 6 is an acquisition node (and no equipotent node has been asserted), we have traversed the RT and generated a single complete overlay that delivers 1-resilience.

## 4.3 Assessing Logical Overlays

Once the set of $k$-resilient logical overlays has been computed, we assess them to determine which ones lead to the longest estimated functional lifetime for the QEP, assuming that node failures are caused by energy depletion alone. This is done by computing the energy drain on each node resulting from the execution of one agenda cycle using the energy cost model in [1]. Given the statistics about current energy levels on each node, we can estimate which node will fail first due to energy depletion. As the agenda cycle repeats at fixed time intervals, we estimate the lifetime of the QEP up to the first node failure. In this way, assuming the resilience afforded by the replication of fragments over the physical nodes in each logical node, we can estimate the lifetime of the QEP over a sequence of node failures. We select the logical overlay that leads to the best estimated lifetime for the QEP over a sequence of failures caused by energy depletion.

## 4.4 Replicating QEP Fragments

The last compile-time step is to assign the QEP fragment in the active node of a logical node to all the $k$ equipotent physical nodes in it. These nodes start in the inactive state, only the node in the original RT is made active for the start of QEP execution. This results in the assignments depicted in Fig. 5(c).

## 4.5 Adapting to Run-Time Node Failure

At run-time, when an active node $a$ fails, if $a$ has a set $I$ of equipotent inactive, non-failed nodes, then the adaptation strategy selects amongst the node $i \in I$ with the smallest average cost for communicating with $lp(a)$ and $lc(a)$. Then, $i$ is removed from $I$ and becomes the active node, $a$ is added to the set of failed nodes, and the children of $a$ are reassigned to become the children of $i$ (i.e., their parent edges are redirected from $a$ to $i$).

At run-time, this adaptive response leads to the following types of messages being sent and responded to: (a) redirection messages, which change the destination node for a node's messages, and (b) activation messages, which cause a node to start executing their agenda. In our example, this means that, at runtime, if Node 1 fails, since $I = \{3\}$, Node 3 receives an activation message and becomes the active node (and $I = \{\}$). Also, Node 2 receives a redirection message and starts sending tuples to Node 3. As a result, the actively executing DAF for the query becomes the one depicted in Fig. 4(b).

Notice that our algorithm does not require the formation of clusters. Some $k$-resilient approaches (e.g., [5]) generate a hierarchy of $k$-sized clusters with a view to distributing the energy expenditure evenly across the levels of the hierarchy. Nodes in a cluster then transmit to a cluster head with a reduced energy cost and the cluster head expends more energy in communicating with other cluster heads or the base station directly. In contrast, we do not use hierarchies, and only one physical node in any logical node is active at any point in time. Other cluster-based algorithms (e.g., [13]) rotate the cluster head periodically in an attempt reduce the likelihood that the node playing that role fails. This is because when the cluster head fails, a new one needs to be elected, which incurs an overhead. In contrast, our approach makes the relevant decisions at compile-time and only incurs the run-time cost of simpler, local, one-way activation and redirection messages.

## 5. EXPERIMENTAL EVALUATION

This section presents some experimental evidence we have collected and that support the hypothesis that the benefit of adapting with a view to increasing the QEP lifetime (and, therefore, the amount of data produced) outweighs the cost incurred in adapting.

The experiments we present here focused on investigating the effect of a sequence of nodes failures throughout the lifetime of a QEP. The goal was to explore the extent to which the logical overlay approach is able to provide resilience to failure and hence yield more tuples/data from the sequence of repaired QEPs than the initial QEP in the sequence is capable of doing.

Throughout the experiments, we adopt a set of assumptions, as follows. We assume that all the metadata has been collected and are accurate and, therefore, we exclude the collection time from our measurements. We assume the SNEE energy and time cost estimation models are accurate as shown in [1]. We assume that our cardinality estimation model (which was thoroughly experimentally validated using the Avrora [12] emulator) is also accurate. We assume that execution starts with fully charged batteries, but our contributions hold even if, from the start, the energy stock differs across different nodes. We assume that an autonomic component that runs outside the WSN on very-capable hardware is in charge of deciding and enacting adaptive responses and, therefore, we exclude the time taken to generate the adaptations from our time measurements.

The aim of the experiments based on a sequence of node failures was to determine if the decisions made in constructing and making use of logical overlays might be cost-ineffective over the sequence of failures with respect to the aggregated lifetime and the number of tuples delivered over the latter.

## 5.1 Experimental Procedure

We began by generating a set of 30 synthetic WSN topologies, with corresponding physical and logical schemas. Each topology comprises 30 nodes with random connectivity to a subset of other nodes.[1] The deployment layout was controlled to give rise to logical overlays with a guaranteed $k$-resilience level, $k = 1$. For a third of the topologies, `SELECT *` queries were generated over the available sensed extent. For another third of the topologies, aggregation queries were

[1]We ran experiments over larger topologies, but no new conclusions emerged so we have omitted the results to save space.

generated. For the final third, join queries were generated. In all cases, we have set the QoS expectations as follows: the acquisition rate is a tuple every 10 seconds and the maximum delivery time is 600 seconds.

For each topology, query, logical and physical schemas, we used SNEE generated the corresponding, initial QEP. We then estimated its lifetime and used this estimate to calculate how many agenda cycles could be executed between node failures so as to uniformly distribute the failures over the lifetime of the initial QEP. We control for which nodes fail after each adaptation until the logical overlay itself fails to allow further adaptive responses.

When simulating the failure of a node, we take the following into account: (a) we prefer confluence nodes both because they are, by definition, crucial for the correctness of the query, and because they tend to have a higher workload than other types of node and are, therefore, more likely to fail in practice; and (b) we exclude acquisition nodes because it allows us to evaluate the net benefit of adaptive responses between functionally-equivalent plans.

In computing costs, we take into account the OTAP costs of disseminating the initial QEP. We also take into account the energy spent executing the QEP during the agenda cycles between failures. This allows us to estimate the functional lifetime of a QEP and express it in terms of the number of executions that an adapted QEP would execute before the next node fails. From this, we can determine the number of tuples delivered through the lifetime of the QEP using our cardinality estimation model (which returns an estimate of the number of tuples returned per agenda execution cycle). Finally, we, of course, take into account the energy and time taken to adaptively respond to each node failure.

## 5.2 Results and Observations

### *Added Lifetime.*

The first experiment fixes one topology and three queries (from each of the classical kinds) and measures the number of agenda cycles that each QEP executes (starting from the initial one, before any node failure, through several adapted QEPs resulting from responses to successive node failures). The experiment compares the results using the logical overlay strategy with those obtained with the original plan. The results are shown in Fig. 6.

The following observations can be made. (1) As successive nodes fail, the estimated lifetime of the successor QEP using logical overlays can grow to twice as large as that of the original QEP. This is because, as each node fails, its replacement starts operating with more energy than the failed node had when it failed therefore boosting the estimated added lifetime. On the negative side, beyond a certain number of failures, there are fewer opportunities to adapt and fewer benefits from adapting (in this experiment) because the scarcity of equipotent nodes begins to restrict what alternative routes can be found. (2) As successive nodes fail, the increase in lifetime can be partly explained by the reduced number of tuples that need to be transmitted. (3) When adaptations do not increase the lifetime, the reason is that the set of nodes which failed did not include the node in the initial QEP that had the most costly agenda execution cost. (4) The join query is a good example of significant benefits ensuing from adaptive responses. The boost in lifetime for the third, fourth and fifth node failures results from the fact that the nodes that fail have the worst estimated lifetimes (because they execute a join and bear large a communication workload). The physical nodes that replace them (as a result of the logical overlay strategy) have significantly longer battery life and, hence, estimated lifetime.

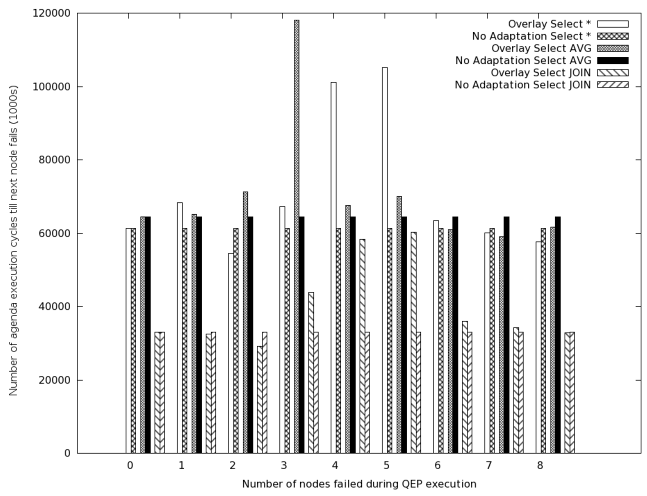


**Figure 6: Additional Agendas Cycles Over a Sequence of Node Failures**

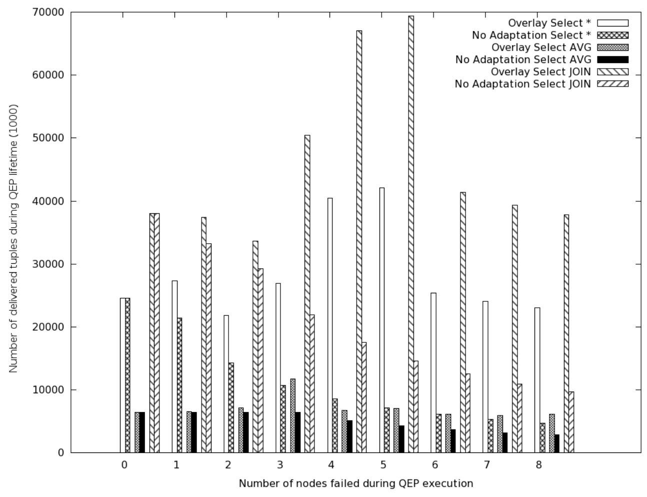


**Figure 7: Tuples Delivered to the Base Station Over a Series of QEPs From Several Node Failures**

### *Added Tuples.*

The second experiment again fixes one topology and three queries (from each of the classical kinds) and measures the number of tuples delivered as each QEP executes (starting from the initial one, before any node failure, through several adapted QEPs resulting from responses to successive node failures). The experiment compares the results using the logical overlay strategy with those obtained with the original QEP. The results are shown in Fig. 7.

The following observations can be made. (1) In all cases, there are benefits in terms of added delivered tuples that ensue from adapting using the logical overlay strategy. (2) As successive, the original QEP loses tuples from acquisition nodes located below the failed node in the routing tree, resulting in less tuples eventually reaching the base-station. (3) As the logical overlay strategy is used to adapt to successive node failures, the adapted QEP enhances the likelihood that tuples from all the acquisition nodes reach the base station thereby leading to a larger number of delivered tuples. (4) The join query is again a good example of significant benefits ensuing from adaptive responses. Note that the third node to fail is a confluence node. As a result the QEP would not emit tuples. By adapting, even though the new QEP executes for less agenda cycles that would otherwise be the case, more tuples are delivered.

## 6. CONCLUSION

SNQPs construe a WSN as a distributed, continuous query platform where the streams of sensed values constitute the logical extents of interest. Crucially, in the context of this paper, they must make assumptions about the connectivity graph of the WSN at compile time that are likely not to hold for the lifetime of the compiled QEP the SNQPs generate. When nodes fail (either due to energy depletion or hardware faults), the connectivity graph changes. From a computational viewpoint, this implies a change in the interconnect that links the distributed elements of the computation taking place in the WSN. The risk in this case is that the overall computation may collapse unless it is capable of adapting to the new interconnect, i.e., the new WSN state. This paper has addressed the problem of extending the lifetime of an evaluating QEP in the event of node failures and has described the following contributions: (1) We have described the logical overlay strategy that enables $k$-resilience to successive node failures in the SNEE SNQP system. (2) We have provided experimental evidence showing that adapting to node failure is beneficial in extending the lifetime of QEPs and in increasing the number of tuples that are delivered to the base station over the lifetime of the QEP.

In future work, we are planning to address the problem of computing a sequence of timed adaptations at compile time so that, proactively and preventively, we take into account the estimated lifetime of a node and reorganize the computation before the node is predicted to fail.

**Acknowledgement** Alan B. Stokes is funded by a doctoral training grant from the UK Engineering and Physical Sciences Research Council and we are grateful for this support.

## 7. REFERENCES

[1] C. Y. A. Brenninkmeijer, I. Galpin, A. A. A. Fernandes, and N. W. Paton. Validated cost models for sensor network queries. In *DMSN*, 2009.

[2] CrossBow. Telosb mote platform. `http://www.willow.co.uk/TelosB_Datasheet.pdf`.

[3] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN*, pages 455–462, 2004.

[4] I. Galpin, C. Y. A. Brenninkmeijer, A. J. G. Gray, F. Jabeen, A. A. A. Fernandes, and N. W. Paton. SNEE: a query processor for wireless sensor networks. *Distributed and Parallel Databases*, 29(1-2):31–85, Nov. 2011.

[5] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *HICSS*, 2000.

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *ASPLOS*, pages 93–104, 2000.

[7] K. Holger and A. Willig. *Protocols and Architectures for Wireless Sensor Networks.* John Wiley and Sons, June 2005.

[8] D. Klan, M. Karnstedt, K. Hose, L. Ribe-Baumann, and K.-U. Sattler. Stream engines meet wireless sensor networks: cost-based planning and processing of complex queries in anduin. *Distributed and Parallel Databases*, 29(1-2), 2011.

[9] M. Liu, S. R. Mihaylov, Z. Bao, M. Jacob, Z. G. Ives, B. T. Loo, and S. Guha. Smartcis: integrating digital and physical environments. *SIGMOD Rec.*, 39(1):48–53, Sept. 2010.

[10] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[11] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Commun. ACM*, 43(5):51–58, 2000.

[12] B. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *IPSN*, pages 477–482, 2005.

[13] M. Ye, C. Li, G. Chen, and J. Wu. Eecs: an energy efficient clustering scheme in wireless sensor networks. In *IPCCC*, pages 535–540, 2005.