The Tenet Architecture for Tiered Sensor Networks

JEONGYEUP PAEK*,
BEN GREENSTEIN‡,
OMPRAKASH GNAWALI*,
KI-YOUNG JANG*,
AUGUST JOKI†,
MARCOS VIEIRA*,
JOHN HICKS†,
DEBORAH ESTRIN†,
RAMESH GOVINDAN*,
EDDIE KOHLER†
*Embedded Networks Laboratory, University of Southern California
‡Intel Labs Seattle
*Stanford University
†Center for Embedded Networked Sensing, University of California, Los Angeles

Author's address:
*Embedded Networks Laboratory, University of Southern California, 3710 S. McClintock Avenue, Ronald Tutor Hall (RTH) 418 Los Angeles, CA 90089; email: {jpeak, kjang, mvieira, ramesh}@usc.edu
‡Intel Labs Seattle, 1100 NE 45th Street, 6th Floor Seattle, WA 98105 email: benjamin.m.greenstein@intel.com
*Stanford University, Clark Center, S255, 318 Campus Drive, Stanford, CA 94305, email: gnawali@cs.stanford.edu
†Center for Embedded Networked Sensing, University of California, Los Angeles, 3563 Boelter Hall Los Angeles, CA 90095-1596; email: {august, kohler, destrin}@cs.ucla.edu, johnhicks@gmail.com
[1] **An earlier version of this article appeared in the proceedings of the ACM Sensys 2006. This manuscript differs from that published version in many ways: it describes a completely re-designed task library, reports revised performance results for tasklets, and discusses experiences from two moderately long-term deployments.**
This material is based in part upon work supported by the National Science Foundation under Grant Nos. 0121778 (The Center for Embedded Networked Systems), and 0520235 (Tenet: An Architecture for Tiered Embedded Networks). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Most sensor network research and software design has been guided by an architectural principle that permits multinode data fusion on small-form-factor, resource-poor nodes, or *motes*. While we were among the earliest promoters of this approach, through experience we found that this principle leads to fragile and unmanageable systems and explore an alternative. The *Tenet architecture* is motivated by the observation that future large-scale sensor network deployments will be *tiered*, consisting of motes in the lower tier and *masters*, relatively unconstrained 32-bit platform nodes, in the upper tier. Tenet constrains multinode fusion to the master tier while allowing motes to process locally-generated sensor data. This simplifies application development and allows mote-tier software to be reused. Applications running on masters *task* motes by composing task descriptions from a novel tasklet library. Our Tenet implementation also contains a robust and scalable networking subsystem for disseminating tasks and reliably delivering responses. We show that a Tenet pursuit-evasion application exhibits performance comparable to a mote-native implementation while being considerably more compact. We also present two real-world deployments of Tenet system: a structural vibration monitoring application at Vincent Thomas Bridge and an imaging-based habitat monitoring application at James Reserve, and show that tiered architecture scales network capacity and allows reliable delivery of high rate data. [1]

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.2.13 [**Software**]: Reusable Software

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Sensor networks, network architecture, tiered network, motes

1. INTRODUCTION

Research in sensor networks has implicitly assumed an architectural principle that, in order to conserve energy, it is necessary to perform application-specific or multinode data fusion on resource-constrained sensor nodes as close to the data sources as possible. Like active networks [Tennenhouse and Wetherall 1996], this allows arbitrary application logic to be placed in any network node. As its first proposers put it:

> **Application-Specific.** Traditional networks are designed to accommodate a wide variety of applications. We believe it is reasonable to assume that sensor networks can be tailored to the sensing task at hand. In particular, this means that intermediate nodes can perform application-specific data aggregation and caching, or informed forwarding of requests for data. This is in contrast to routers that facilitate node-to-node packet switching in traditional networks. [Estrin et al. 1999, Section 2]

This principle has governed the design of data-centric routing and storage schemes [Intanagonwiwat et al. 2000; Ratnasamy et al. 2002], sensor network databases [Madden et al. 2002], programming environments that promote active sensor networks [Levis et al. 2005], and a flexible yet narrow-waisted networking stack [Polastre et al. 2005].

The principle can minimize communication overhead, but at what cost? In our experience, the major costs are increased system complexity and reduced manageability. Systems are hard to develop and debug since application writers are expected to implement sophisticated application-specific routing schemes, and algorithms for multinode fusion, while contending with mote-tier resource constraints.

Adherence to this principle is the main reason why there is currently a significant dichotomy between sensor network deployments and research systems: many of our existing long-lived deployments are relatively simple data collection systems that incorporate no multinode data fusion.

We examine here a different point in the space of possible architectures, motivated by a property common to many recent sensor network deployments [Guy et al. 2006; Szewczyk et al. 2004; Arora et al. 2005]. These deployments have two *tiers*: a lower tier consisting of motes[2] , which enable flexible deployment of dense instrumentation, and an upper tier containing fewer, relatively less constrained 32-bit nodes with higher-bandwidth radios, which we call *masters*. Tiers are fundamental to scaling sensor network size and spatial extent, since the masters collectively have greater network capacity and larger spatial reach than a flat (non-tiered) field of motes. Furthermore, masters can be and usually are engineered to have significant sources of energy (a solar panel and/or heavy-duty rechargeable batteries). For these reasons, most future large-scale sensor network deployments will be tiered. [3]

Future systems could take advantage of the master tier to increase system manageability and reduce complexity, but simply porting existing software to a tiered network would only partially realize these gains. We seek instead to aggressively define a software architecture for tiered embedded networks, one that *prevents* practices that we believe reduce manageability, even at the cost of increased communication overhead.

We therefore constrain the placement of application functionality according to the following **Tenet Principle**: *Multi-node data fusion functionality and multinode application logic should be implemented only in the master tier. The cost and complexity of implementing this functionality in a fully distributed fashion on motes outweighs the performance benefits of doing so.* Since the computation and storage capabilities of masters are likely to be at least an order of magnitude higher than the motes at any point in the technology curve, masters are the more natural candidates for data fusion. The principle does allow motes to process *locally*-generated sensor data, since this avoids complexity and can result in significant communication energy savings. This architectural constraint could be relaxed if required, of course, but aggressively enforcing it most clearly demonstrates the costs and benefits of our approach.

The central thesis of this article is that the Tenet architectural principle simplifies application development and results in a generic mote-tier networking subsystem that can be reused for a variety of applications, all without significant loss of overall system efficiency. Our primary intellectual contribution is the design, implementation, and evaluation of a *Tenet architecture* that validates this thesis.

In Tenet, applications run on one or more master nodes and *task* motes to sense and locally process data. Conceptually, a task is a small program written in a

[2]In this article, we use the term mote broadly to denote a class of sensing devices that for reasons of power, form factor, or price have constrained processing, memory, and communication resources. Such devices are thus incapable of efficiently providing the flexibility, visibility, and robustness of PC-class embedded devices.

[3]Tiering does not imply physical clustering. In the most general definition of a tiered network, a mote can communicate with more than one master, possibly over multiple mote-hops. However, a mote may not have multihop connectivity to all masters in the network.

constrained language. The results of tasks are delivered by the Tenet system to the application program. This program can then fuse the returned results and retask the motes or trigger other sensing modalities. More than one application can run concurrently on Tenet. Our Tenet system adheres to the Tenet architectural principle by constraining mote functionality. All communication to the mote-tier consists of tasks, and all communication from the mote-tier consists of task responses (such as sensor data) destined for a master, so applications simply cannot express mote-tier multinode fusion computation.

Tenet has several novel components. Using our simple yet expressive tasking language, applications specify a task as a linear dataflow program consisting of a sequence of *tasklets*. For example, an application that wants to be notified when the temperature at any mote exceeds 50°F would write a task of the following form.

```
Sample(1000ms, 1, REPEAT, ADC10, TEMP) -> LEQ(A, TEMP, 50)
    -> DeleteActiveTaskIf(A) -> Send()
```

A *task library* implements a collection of composable tasklets. A reliable multitier *task dissemination protocol* ensures highly robust, rapid delivery of tasks from the master tier to the motes. Since different applications may need different levels of reliability for transmitting data back to the application, Tenet implements three qualitatively different data transport mechanisms. Finally, a *routing subsystem* ensures robust connectivity between the tiers; it creates routing table entries in a data-driven fashion in order to reduce overhead.

Tenet has been implemented for networks with MicaZ or TelosB in the mote-tier[4] and 32-bit devices such as Stargates or PCs in the master tier. We have implemented a suite of qualitatively different applications on Tenet, ranging from tracking and vibration monitoring to imaging and network management. Each of these applications is tens of lines of code, and requires no modifications to the rest of the Tenet system.

We have extensively experimented with pursuit-evasion, a particularly challenging application for Tenet since existing implementations deeply rely on multinode data fusion for efficiency [Sharp et al. 2005]. In pursuit-evasion, one or more mobile pursuer robots track and attempt to "capture" one or more evaders with the help of a sensor network. We compared a Tenet implementation with a more traditional one which incorporates mote-tier multinode fusion to reduce redundant evader reports [Sharp et al. 2005]; our Tenet implementation exhibits higher accuracy and lower overhead than mote-PEG at the cost of marginally higher evader detection latency.

We have also successfully deployed two qualitatively different Tenet applications in the real world; structural vibration monitoring at Vincent Thomas Bridge, and image-based habitat monitoring at James Reserve. On the Vincent Thomas Bridge, a tiered sensor network running Tenet reliably collected ambient vibration of the structure at a rate which would have been difficult to achieve on a flat network of motes. At James Reserve, image-sensors (*Cyclops* cameras [Rahimi et al. 2005]) were used to monitor and deliver images of animal trap occupancy, simplifying the

[4]Tenet also supports Mica2 and Mica2dot in the mote-tier although less evaluation has been done on those platforms.

lives of biologists. These deployments suggest Tenet is flexible enough to implement a variety of real applications.

Our evaluation of the individual components of Tenet reveals that Tenet's task library can support high task throughput and that its reliable dissemination mechanism is fast. More generally, our evaluation supports the thesis that the Tenet architectural principle can simplify application development and promote code reuse without much loss of efficiency.

2. THE TENET PRINCIPLE

The Tenet architectural principle moves aggressively away from prevailing practice and prohibits in-mote multinode sensor data fusion. It constrains the processing of data received from multiple sensors to be placed at the master tier, regardless of whether this processing is specific to an application or can be expressed as a generic fusion service. (It might, for example, be possible to cast beamforming or tracking as a generic service that can be used by many applications.) This has two advantages. First, since masters have more processing and storage resources, applications can perform more sophisticated fusion than is possible with the motes. Second, masters can use their high-capacity network to exchange data spanning a large spatial extent, giving fusion algorithms more context into sensed phenomena and resulting in more accurate decisions.

The trade-off is, of course, a potential loss of efficiency which comes in three forms. The first is the opportunity cost of in-mote multinode fusion. For most applications that we have encountered, there is significant *temporal* redundancy, but little *spatial* redundancy (in almost all deployments, we undersample spatially). Since Tenet allows motes to process *locally-generated* sensor data (this local processing is crucial, since it is clearly infeasible to collect time-series data from every sensor in a large sensor network), applications can remove temporal redundancy and recover most of the gains from in-mote multinode fusion. Another, smaller, cost is that processed sensor data needs to be routed over multiple hops from a mote to the nearest master. We argue that even this cost is negligible, since the network diameter in a well-designed sensor network will be small; wireless loss rates being what they are, large diameter sensor networks will be inefficient [Gupta 2000]. The third cost is that in-mote fusion can reduce congestion inside the network. For instance, in pursuit-evasion, leader election within a one-hop neighborhood results in a single evader detection being transmitted; since Tenet does not permit such fusion, it can potentially incur the bandwidth cost of transmitting multiple detections to masters. However, as we show in Section 5, Tenet can avoid this by adaptively setting task parameters so that only nodes having a high-confidence evader detection need transmit their values to a master.

Will the Tenet principle hold when mote processing power, memory, and communication capabilities evolve significantly? The principle simplifies the management and control of constrained motes, and we expect that, for reasons of price, form factor, or battery capacity, motes will continue to be impoverished, *relative to masters*, for the foreseeable future. In absolute terms, motes will become more capable computationally, and possibly with respect to memory. However, in absolute terms, at least in the last few years, we have not seen concomitant growth in battery life

or wireless communication capability. Thus, we contend that this architectural separation will continue to make sense, since otherwise programmers will have to deal with communication and battery life constraints. Moreover, as technology evolves, we will also want to do more with sensor networks (e.g., sense more complex signals, etc.), and we see these relative constraints as continuing to dictate the architectural separation. Even if the technology were to evolve to the point where mote constraints did not matter, our Tenet implementation would still be a viable, flexible programming system for a distributed system of sensors, as our implementations of different applications show.

3. RELATED WORK

In the earliest work on sensor network architecture, Estrin et al. [1999] motivate the need for application-specific multinode aggregation within the network. More recently, Culler et al. [2005] describe SNA, a software architecture that describes the principles by which mote software and services are arranged. They also define the "narrow waist" of the architecture to be a translucent Sensor Protocol layer that exports neighbor management and a message pool, on top of which several network protocols can be built [Polastre et al. 2005]. Chameleon [Dunkels et al. 2007] goes a step further and provides a set of communication primitives that can accommodate variety of underlying protocols and mechanisms. Recently, Essentia [He et al. 2008] proposed "asymmetric function placement", similar in spirit with Tenet, as a guiding principle to architect sensor network systems. It advocates that any nonessential functions should be implemented outside the sensor network to overcome the resource constraints of sensor nodes. Tenet is complementary to this body of work, since Tenet constrains the placement of application functionality in a tiered system and does not address the modularization of software, the lower-layer communication primitives, or the placement of application-independent functionality. The mote-tier in Tenet can be implemented using SP or Chameleon.

The Tenet principle shares some similarities with the Internet's end-to-end principle [Saltzer et al. 1984]. Both principles discuss the placement of functionality within a network, and in both cases the rationale for the principle lies in the tradeoff between the performance advantages obtained by embedding application-specific functionality and the cost and complexity of doing so. However, the Tenet principle is not subsumed by, nor a corollary of, the end-to-end principle, since it is based on a specific technological trend (tiered networks).

The Tenet principle shares some similarities with Active Networks [Tennenhouse and Wetherall 1996]. Specifically, they are similar in that packets (called "capsules" in Active Networks) contain code that are distributed and executed in the network to process application-specific data. However, they differ because Active Networks do not limit application data fusion, and places few, if any, constraints on what can be executed within the network.

Many sensor network deployments in the recent past have been tiered. Examples include the Great Duck Island deployment [Szewczyk et al. 2004], the James Reserve Extensible Sensing System [Guy et al. 2006], and the Extreme Scaling Network [Arora et al. 2005]. In these deployments, tiering provides greater spatial scale and increased network capacity. Other systems have been built upon tiered net-

works. SONGS [Liu and Zhao 2005] focuses on a set of high-level services designed to extract semantic information from a tiered network. Lynch et al. [2003] discuss tiered networks for structural monitoring since upper-tier nodes can perform the sophisticated signal processing functions required for the application. Rhee et al. [2004] describe the design of a tiered network for increasing sensor network lifetime. Tenet, however, is an architecture for building applications for such networks quickly and effectively.

Several pieces of work and have analytically examined the benefits of tiered systems. Liu and Towsley [2003] show that if the number of masters exceeds the square root of the number of motes, a tiered network exhibits no capacity constraint. Mhatre et al. [2005] describe a similar result for the lifetime of a tiered network. Finally, Yarvis et al. [2005] explore how the geometry of tiered deployments affects their lifetime and capacity. This line of research sheds some light on tiered network placement and provisioning.

Many of Tenet's components are inspired by sensor network research over the last seven years. We now discuss these.

Mate [Levis and Culler 2002; Levis et al. 2005] provides a framework for implementing high-level application-specific virtual machines on motes and for disseminating bytecode. A key observation of this work is that a fairly complicated action, such as transmitting a message over the radio, can be represented as a single bytecode instruction provided by an application-specific instruction set. Such an approach can greatly reduce the overhead in disseminating new applications and can simplify application construction. The trade off—as in Tenet—is expressiveness. To the extent that it disseminates and executes task instructions, Tenet defines a virtual machine. However, our focus has been less on the mechanics of bytecode dissemination and execution, and more on defining the right high-level instructions for sensor networks to carry out. ASVMs execute one high-level program at a time, although this program can contain multiple threads. Tenet motes explicitly support concurrent and independent application-level tasks. In this respect, Tenet also differs from tasks in SHARE [Liu et al. 2005], a system that allows applications on a wireless mesh network to express computation in the form of tasks, and optimizes task execution to eliminate or reduce repeated execution of overlapping task elements.

Tenet's mote software runs with the TinyOS [Hill et al. 2000] operating system and is written in nesC [Gay et al. 2003]. TinyOS' design choices (pushing as many decisions as possible to compile-time) have led to the development of an impressively stable and flexible runtime and library of drivers. However, static allocation is not the right model for dynamically invoked application-level tasks. Although our software runs on TinyOS, it makes widespread use of dynamic memory and function pointers. Tenet's attribute-based data structures and processing chains are similar to Snack [Greenstein et al. 2004]. Also, Tenet's tasking language has some resemblance to that of DSN [Chu et al. 2007]. However, DSN is intended for programming the lowlevel behavior of individual motes whereas Tenet's tasking language is used to express application-level tasks that can be dynamically disseminated and executed at runtime.

Tenet, like many proposed macroprogramming techniques such as *MacroLab* [Hnat

et al. 2008], *Pleiades* [Kothari et al. 2007], and *Regiment* [Newton et al. 2007], attempts to provide a general-purpose programming framework that relieves the complexity of distributed programming. However, Tenet achieves this goal by constraining mote-tier functionality, while macroprogramming approaches, in general, rely on compiler and runtime technology to provide higher-level programming abstractions.

To our knowledge, no prior work has proposed or implemented a complete networking subsystem for a tiered network, as we have. The components of the networking subsystem bear some resemblance to prior work, but are uniquely influenced by Tenet's communication patterns and generality.

TRD (Tiered Reliable Dissemination, Section 4.3.2) employs similar techniques (exponential timers and suppression) as prior code dissemination protocols [Levis et al. 2004; Stathopoulos et al. 2003; Hui and Culler 2004]. However, although all reliable dissemination mechanisms employed for code dissemination assume a single sender (for obvious reasons), TRD supports multiple masters sending different tasks concurrently to all motes, and employs reliable flooding both on the master and the mote-tiers. Second, reliable code dissemination designs are optimized for the particular application; unlike TRD's summaries, the metadata summaries used for loss recovery are specific to code pages.

Reliable data transport has received some attention in the literature. RMST [Stann and Heidemann 2003] (Reliable Multi-Segment Transport) adds reliable transport on top of Directed Diffusion. RMST is a NACK-based protocol in which loss is repaired hop-by-hop. However, unlike Tenet's transport, it is tightly integrated with Diffusion, designed for larger and more capable platforms, and optimized for recovering losses of image fragments. PSFQ [Wan et al. 2002] (Pump Slowly, Fetch Quickly) is a hop-by-hop reliable transport protocol designed for sensor network reprogramming, Wisden [Xu et al. 2004] is a system for reliably transporting structural vibration data from a collection of sensors to a base station, and DataRel [Stathopoulos et al. 2005] provides a TCP-like abstraction for transporting data from a mote to a border master in a tiered network. Finally, Flush [Kim et al. 2007] is a reliable bulk transport protocol that reduces transfer time by carefully controlling the rate at each hop along a flow, assuming that data collection is coordinated to allow only one flow in the network at a time. In contrast to these systems, Tenet's reliable transport ensures point-to-point delivery of processed sensor data from a mote to any master in a tiered network.

Most prior routing protocols either support data delivery to a base station [Polastre et al. 2005; Woo and Culler 2003] or to multiple sinks [Guy et al. 2006]. Some, such as Centroute [Stathopoulos et al. 2005], also centrally compute efficient source routes to individual motes on demand, supporting unicast traffic from a base station to a mote. Tenet's routing system, in contrast, supports unicast traffic from a mote to any master in the tiered network, and builds routing entries for traffic in the reverse direction in a data-driven fashion.

4. THE TENET SYSTEM

In this section, we first describe a set of design principles that guided the development of the Tenet implementation. We then describe, in some detail, the two

main components of Tenet, its *tasking subsystem* and its *networking subsystem*. We conclude with a discussion of the limitations of our current Tenet prototype, motivating directions for future work.

4.1 Design Principles

The Tenet architectural principle constrains the design space for sensor network architectures, but is not itself an architecture. Our Tenet system is based on the Tenet principle and the five additional *design principles* described here. Again, we define these principles aggressively, since this will show when violating the principles is necessary for network performance or functionality.

The Tenet principle prohibits multinode fusion in the mote-tier. The precise form of this prohibition is expressed as restriction on sensor network communication, which must take the form of **Asymmetric Task Communication:** *Any and all communication from a master to a mote takes the form of a task. Any and all communication from a mote is a response to a task; motes cannot initiate tasks themselves.* Here, a "task" is a request to perform some activity, perhaps based on local sensor values; tasks and responses are semantically disjoint. Thus, motes never communicate with (send sensor data explicitly directed to) another mote. Rather, masters communicate with (send tasks to and receive data from) motes, and vice versa.

The second principle expresses the **Addressability** properties of a Tenet network: *Any master can communicate with any other master as long as there is (possibly multihop) physical-layer connectivity between them; any master can task any mote as long as there is (possibly multihop) physical-layer connectivity between them; and any mote should always be able to send a task response to the tasking master.* This principle helps enforce high network robustness and a relatively simple programming model. For example, imagine that a mote A is connected one-hop to a master M, but could be connected to a different master, M', via three hops. The addressability principle requires that, if M fails, A will learn about M' and be able to send responses via M', and vice versa. The requirement to support master-to-master communication allows, but does not require, the construction of distributed applications on the masters. Addressability requires much less of motes, however; a mote must be able to communicate with at least one master (assuming the network is not partitioned), not all masters, and mote-to-mote connectivity is not required. This is by design, and greatly simplifies mote implementations.

The third **Task Library** principle further defines what tasks may request of a mote: *Motes provide a limited library of generic functionality, such as timers, sensors, thresholding, data compression, and other forms of simple signal processing. Each task activates a simple subset of these functionality.* A task library that simultaneously simplifies mote, master, and application programming while providing good efficiency is a key piece of the Tenet architecture. We discuss the design of the task library shortly.

Finally, **Robustness** and **Manageability** are primary design goals. Robust networking mechanisms, which permit application operation even in the face of extensive failures and unexpected failure modes, are particularly important for the challenging environments in which sensor networks are deployed. Manageability implies, for example, that tools in the task library must provide useful insight

into network problems (such as why a particular sensor or group of sensors is not responding, or why node energy resources have been depleted far faster than one would have expected) and allow automated response to such problems.

Three important advantages arise from these design principles. First, applications execute on the master tier, where programmers can use familiar programming interfaces (compiled, interpreted, visual) and different programming paradigms (functional, declarative, procedural) since this tier is relatively less constrained. Second, the mote-tier networking functionality is generic, since Tenet's networking subsystem merely needs to robustly disseminate task descriptions to motes and reliably return results to masters. This enables significant code reuse across applications. Finally, mote functionality is limited to executing tasks and returning responses, enabling energy-efficient operation.

4.2 Tasks and the Task Library

When developing a language for describing tasks, the key trade-off faced is between expressiveness and simplicity. Conventional Turing-complete languages place few or no restrictions on what a programmer may request a mote to do, but make tasks error prone, hard to understand, and hard to reuse.

Tenet chooses simplicity instead. A Tenet task is composed of arbitrarily many *tasklets* linked together in a linear chain. Each tasklet may be thought of as a service to be carried out as part of the task; tasklets expose parameters to control this service. For example, to construct a task that samples a particular ADC channel every 500 ms and sends groups of twenty samples to its master with the tag `LIGHT`, we write

```
Sample(500ms, 20, REPEAT, ADC0, LIGHT) -> -> Send()
```

or equivalently,

```
Repeat(500ms) -> Sample(ADC0, LIGHT) -> Pack(LIGHT, 20) -> Send()
```

Tenet restricts tasks to linear compositions of tasklets for simplicity and ease of construction and analysis, another example of an aggressive constraint. Tasks have no conditional branches or loops, although certain tasklets provide limited versions of this functionality. For example, a tasklet within a task may repeat its operation. Also, a tasklet may, depending on its parameters, ignore or delete certain inputs or terminate its execution, implementing a constrained conditional.

Tenet's task library was inspired by our own prior work on SNACK [Greenstein et al. 2004] and VanGo [Greenstein et al. 2006], and somewhat resembles other sensor network tasking architectures [Liu et al. 2005]. It can also be seen as a particular instantiation of a virtual machine [Levis and Culler 2002]. However, rather than allow application-specific VMs, we have worked to make this VM flexible, general, high level, and efficient enough to support easy programming and tasking from masters.

4.2.1 *Task Building Blocks.* The tasklets in the Tenet task library provide the building blocks for a wide array of data acquisition, processing, filtering, measurement, classification, diagnostic, and management tasks. For example, the `Get()` tasklet collects a variety of information from the system such as routing state or statistics on dynamic memory usage, while the `Issue()` tasklet controls when a

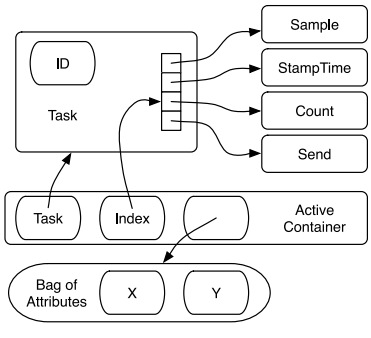


Fig. 1. Mote data structures for a typical task.

task should be initiated either by delaying the subsequent tasklets on a task chain until a specified time or by initiating a task periodically at a given interval.

The difficulty in constructing such a library is in determining what the right building blocks should be. The `Reboot()` tasklet should, of course, reboot the node. But what should `Sample()` do? Should it know how to be periodic? Should it know how to pack several samples into a packet? Also, should we make one `Arith()` tasklet? or should we separate `Add()` and `Subtract()` tasklets?

Our guiding principle is that tasklets should provide as much functionality as possible, while still being easy to use and reuse. So, `Sample()` knows how to repeat and collect blocks of samples because this level of functionality is still easy to control (the parameters to `Sample()` are intuitive) and it is still fairly efficient. (A Tmote mote can manage the state of roughly 100 `Sample()` tasklets even with its limited RAM.) Also, `Arith()` can perform add, subtract, multiply, etc., because it only requires one extra parameter, `OP_TYPE`, while avoiding replication of code.

Our guiding principle can result in tasklets with many parameters. To simplify programming these tasklets, Tenet provides the user with a rich set of easy-to-use master-side tasklet APIs (Appendix A.1). These APIs are translated at the Tenet master into the appropriate tasklets supported at the mote. For example, the `Add()` and `Sub()` master-side tasklets are equivalent to `Arith(ADD)` and `Arith(SUBTRACT)` respectively, and `NextHop()` is identical to `Get(NEXT_HOP)`. This approach incurs no overhead on the motes, since the translation is performed at the task parser on the master. Furthermore, it enables a simpler, more user-friendly, and less error-prone API, and makes it easier to statically analyze tasks on the master-side for parameter validity and usage.

4.2.2 *The Task Data Structure.* Figure 1 presents the task data structures from the perspective of a mote. A task executing on a mote has two parts: a task object maintains the tasklet chain, and one or more *active containers* hold intermediate and final results of task processing.

The task object includes a task ID and a list of tasklets with their corresponding parameters. During installation each tasklet's constructor is called, passing any parameters specified by the master; the constructor returns to the task object these parameters, pointers to methods to run and delete the tasklet, and any needed tasklet state. Finally, a new active container is created that points at the first

tasklet in the chain. The underlying scheduler will soon start task processing by passing this active container to the first tasklet.

An active container corresponds to an instance of a task whose processing is in progress. The active container moves from tasklet to tasklet, being processed by each in turn. Once it reaches the end of the chain of tasklets, it is deleted. An active container maintains a pointer to its task object and an index specifying where it is currently in the task object's linear chain of tasklets. A repeating tasklet clones the active container after each iteration. For example, while executing the task

```
Repeat(1000ms) -> LocalTime(X) -> Send()
```

an active container is cloned and emitted from the `Repeat()` tasklet every 1000 ms. Thus, there may be several active containers associated with the same task in the system at the same time.

An active container also holds a bag of attributes, containing all data resulting from task execution. Each attribute is simply the tuple ⟨*tag, length, value*⟩. For example, in the preceding task, the `LocalTime()` tasklet adds an attribute with the tag `X` to the bag.

In general, tasklets name their input and output data types explicitly using these tags. Thus

```
Sample(10000ms, 1, REPEAT, ADC_LIGHT, X) -> LEQ(Y, X, 39)
    -> DeleteActiveTaskIf(Y) -> LocalTime(Z) -> Send()
```

will send timestamped light values only when their values are greater than 39. (The actual implementation is done by deleting the execution of current instance if the light value is less than or equal to 39). It is up to the task composer to verify that the data type specified as input to an tasklet is compatible with that tasklet.

All state associated with a task is maintained by that task. Tasks and tasklets are dynamically allocated. Since tasks arrive unpredictably, it is impractical to statically plan memory allocation for them; dynamic allocation improves system efficiency by providing a way to allocate memory resources only where needed.

4.2.3 *Mote Runtime.* The Tenet mote runtime provides a set of task-aware queues for waiting on hardware resources. Each queue is owned by a *service* corresponding to a particular tasklet. For example, the `Issue()` tasklet, which implements the `Wait()` and `Repeat()` API's, delays a task for a period of time. Its corresponding time *service* maintains a queue where tasks may reside until it is time for them to proceed. Likewise, the `Sample()` service maintains a queue of tasks waiting for the ADC resource.[5]

At the heart of the system is the Tenet scheduler. It maintains a queue of tasks waiting to use the mote's microcontroller. The scheduler operates at the level of tasklets, and knows how to execute the task's tasklets in order. Since it operates at this level of granularity (as opposed to executing each complete task one at a time), several concurrently executing tasks may get fair access to a mote's resources.

[5]Arbitration between tasklets that use the same resource is important. We expect the underlying system (e.g., TinyOS) to provide this functionality.

Tasklets	Description	ROM	RAM
Actuate	Actuate specified actuation channel	272	0
Arith	Arithmetic operation on attribute data	592	0
Attribute	Check existence/non-existence of an attribute	202	0
Bit	Bitwise operation on attribute data	496	0
Comparison	Comparison operation on attribute data	558	0
Count	Increments a counter	122	0
DeleteActiveTaskIf	Deletes an active task, possibly conditional	152	0
DeleteAttributeIf	Deletes attribute(s), possibly conditional	226	0
DeleteTaskIf	Deletes the task, possibly conditional	156	0
Get	Get system state information (GlobalTime, NextHop, etc.)	850	0
Image	Control Cyclops camera and take image	2098	243
Issue	Issue a task after delay (periodic or specified time)	1582	28
Logical	Logical operation on attribute data	402	0
OnsetDetector	Filter data by detecting onset of significant event	1996	0
Pack	Pack consecutive scalars into a vector attribute	292	0
Reboot	Reboots the mote	56	0
SendPkt	Send using packet-based or best-effort transport	3742	398
SendStr	Send using stream-based transport	4442	560
SendRcrt	Send using RCRT protocol	6198	866
Sample	Samples specified ADC channel	7224	146
SampleRssi	Uses radio RSSI as a virtual sensor	226	18
SampleMda400	Samples specialized vibration board	2324	18
Statistics	Statistics of an attribute data	462	0
Storage	Store/Retrieve an attribute, valid across executions of task	544	6
Voltage	Samples battery voltage (code partly overlaps with Sample)	3444	66
UserButton	Trigger/block task execution using user button (Telosb)	360	16

Fig. 2. Tasklets in the Tenet task library, with ROM and RAM bytes saved by removing each tasklet from our mote application.

4.2.4 *Task Operations.* The task description disseminated from a master to its motes contains, in serialized form, a task identifier and the list of tasklets that comprise this task. Each tasklet encodes its name (from a well-known enum) and its parameters as a tag-length-value attribute.

Motes accept two operations on tasks: installation and deletion. When a mote receives a task description from a master containing a task ID that is not currently installed, the mote concludes that this task should be installed. An empty description with currently installed task ID is interpreted as a request to destroy a running task. To delete a task, all active containers corresponding to that task are found and destroyed (they may be hiding in any tasklet's associated service or in the scheduler), followed by the task object itself.

4.2.5 *OS Dependencies.* The Tenet task library was implemented on top of TinyOS, but we follow very few of the programming patterns of that operating system. The advantage to using TinyOS is in the robustness and availability of its drivers. The chief drawback is that we cannot dynamically load software libraries at runtime; thus, all tasklets an application might require must be compiled into a single binary. In the future, we may consider OSes that relax this restriction [Han et al. 2005], allowing required tasklets to be fetched from a master dynamically.

4.2.6 *Examples of Tasks.* Figure 2 describes Tenet's current tasklets and their contributions to program size and static memory allocation. Also, Appendix A.1 lists all of Tenet's tasklet APIs provided by current set of tasklets. We may link together these building blocks to compose a wide array of sensing, maintenance, and diagnostic tasks.

For instance, we have composed several tasks that assess and maintain the health of a sensor network. To verify by visual inspection that a mote is running properly, we may inject the *Blink* task.

```
Repeat(1000ms) -> Count(A, 0, 4) -> SetLeds(A)
```

Every second, this task adds 4 to a counter whose initial value is 0, places the value of this counter into an attribute called A, and displays A as a pattern of LEDs. *CntToLedsAndRoutedRfm* is a more complicated task to help verify that there is end-to-end connectivity from a master to a mote.

```
Repeat(1000ms) -> Count(A, 0, 1) -> SetLeds(A) -> Send()
```

In addition to blinking, this task transmits the value of A back to the master node.

To further diagnose our motes, we may monitor routing table information and the memory usage by issuing *Ping* and *MeasureHeap* tasks.

```
Repeat(1000ms) -> NextHop(A) -> Send()
Repeat(1000ms) -> MemoryStats(B) -> Send()
```

Ping reports the routing table's next hop information every second. *MeasureHeap* reports a mote's statistics on current and peak dynamic memory allocations on the heap. Such tasks, as well as data acquisition and processing tasks, may run concurrently.

If mote software seems to be behaving poorly, the `Reboot()` tasklet may be used to reset a mote. This is often the most prudent recourse. Of course, some mote software failures within the routing, transport, and task installation software may prevent the proper reception, installation, and execution of the *Reboot* task.

The tasklet `Sample()` serves as the data source for acquisition and processing chains. A `Send()`-style tasklet is usually the chain's tail; the particular tasklet depends on the type of transport desired (see the following). For example,

```
Sample(1000ms, 1, REPEAT, 1, ADC0, A) -> Send()
```

provides the most basic sampling and transmission support one might expect from a mote. Every second, this task takes a sample from the ADC's channel 0, gives it the name A, and transmits it. It is similar to the TinyOS *SenseToRfm* application in that it periodically collects a single sensor value, and to TinyOS's *Surge* application in that it delivers data using multihop transport.

Tasklet parameters and their linear composition make it fairly flexible. The following configuration, for example, samples both channels 0 and 1 and stores five samples of each before sending them.

```
Repeat(1000ms) -> Sample(ADC0, A) -> Sample(ADC1, B)
    -> Pack(A, 5) -> Pack(B, 5) -> Send()
```

To instruct a mote to process the samples it collects, a master may specify tasklets between `Sample()` and `Send()`. A rather complicated example is as follows.

```
Sample(1000ms, 10, REPEAT, ADC0, A)
    -> MeanDev(B, A) -> SetLeds(A)
    -> CountGEQ(C, A, 45) -> GEQ(C, C, 3)
    -> DeleteAttributeIf(C, A) -> DeleteAttribute(C)
    -> GlobalTime(D) -> Count(C, 0, 1)
    -> Send()
```

Every second, this task takes a sample from the ADC, waits until 10 samples are collected, and passes this set of ten samples at a time through the task chain. On each pass through the chain, the task measures the mean deviation of the sample set, and displays on the LEDs a pattern representative of the values of the samples. The task also classifies the sample set as interesting if at least three of them have an amplitude of at least 45 (by counting how many of them have an amplitude of at least 45 and checking whether there are at least three of them). It then records the global timestamp and a sequence number, and transmits the sample set along with the measured mean deviation. The sample itself is transmitted as well, but only if it happens to be interesting. Notice that the tasking language allows reuse of attributes. For example, `GEQ(C, C, '3')` uses attribute C as both input and output where the creation of an output attribute will automatically delete any attribute with the same name.

4.2.7 *An Example of a Data Collection Application.* How can a Tenet application make use of this tasking library? Shown shortly is the code for *a complete application* which collects voltage level, next routing hop, and global time information every 5 seconds. We support Tenet application development in C or Python; the pseudocode that follows is slightly abstracted from a C program.

```
int main()
{
    task_string = "Sample(5000ms, 1, REPEAT, 2, VOLTAGE) \\
                    ->NextHop(3)->GlobalTime(4)->Send()";
    task_packet = construct_task(task_string);
    task_id = send_task(task_packet);
    while (1) {
        (response, mode_id) = read_response();
        voltage = response_find(2, response);
        nexthop = response_find(3, response);
        globaltime = response_find(4, response);
        store_data_to_file(mote_id, voltage, nexthop, globaltime);
    }
}
```

The Tenet master-side API enables a programmer to construct a tasking packet from a task description (`construct_task()`), disseminate the task (`send_task()`), receive task responses from the motes (`read_response()`), and find the attribute of interest from this response (`response_find()`). The programmer only needs to write down the task description, extract data from the response, and process the data. In the previous application, the data is stored in a file, but a more realistic application might process the data as it is received.

4.3 The Networking Subsystem

Tenet's networking subsystem has two simple functions: to disseminate tasks to motes and to transport task responses back to masters. The design of the networking subsystem is governed by four requirements.

The subsystem should support different applications on tiered networks. Routing and dissemination mechanisms in the prior sensor network literature do not support tiered networks (Section 3). As a result, we had to build a complete networking

Function	Mechanism
Disseminating tasks from a master to a mote	Tiered reliable flooding of a sequence of packets from any master to all motes
Routing task responses from a mote to a master	Tiered routing, with nearest master selection on the mote-tier, and overlay routing on the master tier
Routing transport acknowledgments from master to mote	Data-driven reverse-path establishment
End-to-end reliable transport of events	Transactional reliable transmission protocol
End-to-end reliable transport of time series	Stream reliable transmission with negative acknowledgments

Fig. 3. Tenet networking mechanism summary.

subsystem from the ground up while leveraging existing mature implementations where possible. An important design goal was to support many application classes, rather than tailoring the networking subsystem to a single class.

Routing must be robust and scalable. The routing system must find a path between a mote and a master if there exists physical multihop connectivity between them. In Tenet, the routing system needs to maintain state for motes since masters may need to transmit packets to motes; the routing state on the motes must, in the worst case, be proportional to the number of actively communicating motes. For routing data back to the masters, the system will need to maintain state for masters. We require that this state be constant, independent of the number of masters. Intuitively, this is the best a tree-based mote routing system can do, without increasing packet header size. By using source routing, as in Centroute [Stathopoulos et al. 2005], it is possible to remove routing state at the motes entirely, and Tenet can support this form of routing as well.

Tasks should be disseminated reliably from any master to all motes. Any master should be able to task motes, so task dissemination must work across tiers. Furthermore, tasks must be disseminated reliably, and a mote must be able to retrieve recently disseminated tasks after recovery from a transient communication or node failure.

Task responses should be transported with end-to-end reliability, if applications so choose. Some applications, such as structural monitoring or imaging, are loss intolerant. Furthermore, as applications push more data processing onto the motes, this processed data will likely be loss intolerant. This makes end-to-end reliable transmission a valuable service Tenet should support. While many existing systems use a limited number of hop-by-hop retransmissions or hop-by-hop custodial transfer (retransmit until the next hop receives the packet), neither of these mechanisms ensures end-to-end reliable delivery; for example, if the receiver of a custodial transfer fails immediately after the transfer is complete, data may be irretrievably lost.

The following sections describe the design of Tenet's networking subsystem and how the design achieves these goals; Figure 3 summarizes its novel mechanisms.

4.3.1 *Tiered Routing.* In Tenet, all nodes (masters and motes) are assigned globally unique 16-bit identifiers. The identifier size is determined by the TinyOS networking stack, and motes use the 16-bit TinyOS node identifier. Masters run IP, and use the lower 16 bits of their IP address as their globally unique identifier for master-to-mote communication. This requires coordinated address assignment between the master and the mote-tiers, but this coordination does not impose

significant overhead during deployment.

Tenet's routing system has several components: one component (master tier routing) leverages existing technology; another component (mote-to-master routing) is adapted from existing tree-routing implementations; and two other components (data-driven route establishment for master-to-mote routing, and overlay routing on the master tier) are novel.

Our addressability principle requires masters to be able to communicate with each other. Tenet simply uses IP routing in the master tier. This has two advantages. First, distributed Tenet applications can use the well-understood socket interface for communicating between distributed application instances. Second, Tenet can leverage routing software developed for wireless IP meshes. Our current implementation allows multiple IP routing mechanisms within the master tier; our experiments used static routing, but Tenet can easily accommodate other wireless routing protocol implementations such as Roofnet [Aguayo et al.] and OLSR [Clausen and Jacquet 2003].

Tenet's addressability also calls for any mote to be able to return a response to the tasking master. Standard tree-routing protocols are inadequate since they assume a single base station. Tenet uses a novel *tiered routing* mechanism, where a mote's response is first routed to its nearest master, and is then routed on the master tier using an overlay. In order to enable motes to discover the nearest master, each master periodically transmits beacons. When a mote receives a beacon, it relays this to its neighbors after updating a path metric reflecting the quality of the path from itself to the corresponding master. Then, the mote selects as its "parent" that neighbor which advertised the best path to a master. Over time, a mote's parent may change if the path quality to its nearest master degrades, or if the nearest master fails, conditions detected by the periodic beacons. This only requires a mote to maintain state for a one master plus a fixed number of potential alternate masters, and as long as a mote can hear at least one master, it can send traffic to the master tier. We have modified three well-known tree routing protocols (MultiHopLQI, MultiHopRssi, and MintRoute)[6] to support this nearest master selection. When a mote receives a packet from any neighbor that is *not* its parent, it forwards the packet to the parent.

Once the packet reaches the master tier, an IP overlay is used to forward the packet to the destination master. A master node that gets a packet from the mote-tier examines the 16-bit destination address on the packet. It translates this to an IP address (by prepending its own 16-bit subnet mask to that address), then determines the next hop towards this IP address using the IP routing table. It then encapsulates the packet in a UDP packet, and sends this to the next hop. The next hop master node repeats these actions, ensuring that the packet reaches the destination. This IP overlay is implemented as a user-space daemon. Together with nearest master selection, IP overlay routing ensures that if there exists a path between a mote and the master to which it should send its task response, that path

[6]We use MultiHopLQI in our PEG experiments (Section 5.2) and VTB deployment (Section 5.4), and MultiHopRssi in our JR deployment (Section 5.5). MultiHopLQI, MultiHopRssi and MintRoute are implemented in TinyOs-1.x. Our TinyOs-2.x port uses modified versions of MLQI and CTP [Gnawali et al. 2009].

will be taken.

The routing system also enables point-to-point routing between masters and motes. This is necessary in two cases. First, our reliable transport mechanisms (described shortly) require connection establishment and acknowledgment messages to be transmitted from a master to a mote. Second, in certain circumstances, it may be necessary to adaptively retask an individual mote; for efficiency, a master can directly send the task description to the corresponding mote instead of using our task dissemination mechanism described soon. In either case, a master needs to be able to unicast a packet to a mote only after it has received at least one packet from the mote.

Tenet's scalable data-driven route establishment mechanism works as follows. When a mote gets a task response data packet from a nonparent, it establishes a route entry to the source address (say S) in the packet, with the next hop set to the sender. It also implements an aging mechanism; the age of a new route entry is set to zero, and when the age exceeds a certain limit (in our implementation, 2 minutes), this entry is removed. If the entry existed previously, the mote resets the associated age. Only after updating the state does it forward the packet to the parent. Subsequently, when the parent sends a packet destined to S (say a transport acknowledgment from a master), the node uses this routing entry to forward the packet to S, and resets the associated age. Thus, the routing entry is active as long as a mote has recently communicated with its master. Masters also implement a similar algorithm that sets up these data-driven routes so packets on the master tier are correctly routed towards S.

Thus, data-driven route establishment maintains one routing entry per actively communicating mote. More precisely, each mote maintains one routing entry for each active mote in its own subtree. Data-driven route establishment can have degraded data delivery performance in the presence of link asymmetry. For this reason, we have recently added support for the CTP routing protocol, to TinyOs-2.x implementation of Tenet, which has larger memory footprint but uses bidirectional ETX as its routing metric. More generally, the Tenet architecture does not restrict the choice of mote-tier tree routing protocol. Hence, asymmetric links are not a challenge specific to Tenet, and any routing solution developed to address this problem can be incorporated into Tenet.

4.3.2 *Tiered Task Dissemination.* A central component of Tenet's networking subsystem is one that disseminates tasks from masters to motes. Tenet's task dissemination subsystem reliably floods task descriptions to all motes. This choice is based on the observation that in most of the applications we describe in this article, and those we can foresee, tasking a network is a relatively infrequent event, and applications usually task most if not all the motes. When applications need to select a subset of the motes to task, they can indicate this by prepending a *predicate tasklet* to the task description. A predicate is a function of static attributes of a node (such as the sensors it has, its current location, and so on). All motes begin executing the task, but task execution is completed only on motes whose attributes satisfy the predicate. For example, to execute a task only on motes whose ID is less than 10, the following tasklet chain can be prepended to the task.

```
NodeId(A) -> GT(B, A, 10) -> DeleteTaskIf(B) -> ...
```

Tenet's reliable task dissemination mechanism is built upon a generic reliable flooding protocol for tiered networks called *TRD (Tiered Reliable Dissemination)*. TRD provides the abstraction of reliably flooding a sequence of packets from any master to all motes in the network. TRD's abstraction is different from that considered in the literature (Section 3). Disseminating a task using TRD is conceptually straightforward. Applications send the task to TRD; if the task description fits within a packet, TRD sends the packet directly, otherwise it fragments the task description into multiple packets which are then reassembled at each mote.

TRD works as follows. Suppose a master M wishes to transmit a task packet. TRD locally caches a copy of the packet on M, assigns a sequence number to the packet, and broadcasts the packet to its neighbors as well as to any nearby motes (in case M happens to have motes nearby). Motes also cache received packets, and rebroadcast previously unseen packets to their neighbors, and so forth. Each cache entry contains <master id, sequence number> tuple along with a copy of the packet and its age since creation. In our implementation, both master and mote caches are of fixed size (25 entries), and cache entries are replaced using an LRU policy.

Of course, some motes or masters may not receive copies of the packet as a result of wireless transmission errors. To recover from these losses, each node (master or mote) occasionally transmits a concise summary of all the packets it has in its cache. These transmissions are governed by an exponentially backed off timer [Levis et al. 2004], so that when the network quiesces, the overhead is minimal. The summaries contain the last k (where k is a parameter determined by the memory available on the motes) <master id, sequence number> tuples, one tuple from each active master id in the cache with the latest sequence number for that master. If the node detects that a neighbor has a newer packet (identified by <master id, sequence number> tuples) than what it has in its own cache, it immediately requests the missing packet using a *unicast* request. If a node detects that a neighbor has some missing packets, it immediately broadcasts a summary so the neighbor can rapidly repair the missing sequence packets, and so other nodes can suppress their own rebroadcast. Lastly, when a node boots up, it broadcasts an empty summary prompting neighbors to send their summaries.

This protocol has several interesting features. It uses the master tier for dissemination; as we show in our experiments, this results in lower task dissemination latencies than if a single master were used to inject tasks into the mote cloud. It is extremely robust: all the nodes in the network would have to simultaneously fail for a packet to be lost. It periodically transmits small generic summaries, proportional in size to the number of active masters.

On the master side, TRD is implemented within a separate transport layer daemon that also implements the transport protocols described next. Applications connect to this daemon through sockets and transmit a task description to TRD (our implementation has a procedural interface `send_task()` that hides this detail from the programmer). Before transmitting a new task, the daemon assigns it a unique task ID and maintains a binding between a task ID and the application. Applications can use this task ID to delete tasks, or to associate task responses to tasks. This ID is also included in task responses so that the transport layer knows

which application to forward the response to. Our TRD implementation checkpoints assigned task IDs to avoid conflicting task ID assignments after a crash.

An interesting side effect of TRD's robustness is that a node which recovers after a crash may receive a copy of a task it had previously received or sent out recently in the past. This may cause problems; for example, if a `Reboot()` task was sent to reset the motes, a mote might repeatedly reboot. To avoid this undesirable situation, every TRD message has an age field which is incremented using a timer and synchronized through summary exchanges. TRD checks the age of the received task message and deletes this task if it is older than the node's lifetime. Hence, only the tasks that are generated after a node boots up will be executed on that node.

Our mote implementation of TRD is engineered to satisfy mote resource constraints and to support multiple platforms. TRD stores its packet cache in Flash memory in order to conserve RAM and only maintains a small index of the Flash contents in RAM. Furthermore, TRD currently only supports a fixed number of masters (currently 5) to limit the size of the index. It implements a consistent aging strategy by which master entries in the summary are individually timed out to accommodate new active masters.

Finally, Tenet applications might also wish to adaptively retask a subset of the motes; for example, an application might choose to adjust the sampling rate on some sensors based on observed activity. Applications can use TRD to delete the previous task and disseminate a modified task description with an appropriate predicate or, if it is more efficient, send the updated task descriptions directly to the motes using one of Tenet's reliable transport protocols described in the next section.

4.3.3 *Reliable Transport.* Tenet needs a mechanism for transmitting task responses from a mote to the master that originated the task, possibly with end-to-end reliability. Tenet currently supports four types of delivery mechanisms: a best-effort transport useful for loss-tolerant periodic low rate applications, a transactional reliable transport for events, a stream transport for high-data rate applications (imaging, acoustics, vibration data), and the RCRT [Paek and Govindan 2007] protocol for congestion-controlled reliable data collection. All four delivery mechanisms use a limited number of hop-by-hop retransmissions to counter the high wireless packet loss rates encountered in practice. Applications select a transport mechanism by using the corresponding tasklet in their task description (respectively, `Send()`, `Send(E2E_ACK)`, `SendStr()`, or `SendRcrt()`). The implementation of best-effort transport is conventional, and that of RCRT protocol is detailed in Paek and Govindan [2007]; the rest of this section discusses the transactional and stream transport mechanisms.

Transactional reliable transport allows a mote to reliably send a single packet (containing an event, for example) to a master. In Tenet, transactional transport is implemented as a special case of stream transport: the data packet is piggybacked on stream connection establishment.

The stream transport abstraction allows a mote to reliably send a stream of packets to a master. When a task invokes the `SendStr()` tasklet, the stream transport module first establishes an end-to-end connection with the corresponding

master. Stream delivery uses a connection establishment mechanism similar to the three-way handshake mechanism of TCP. However, because stream delivery is fundamentally simplex, the connection establishment state machine is not as complex as TCP's and requires fewer handshakes for connection establishment and teardown. Once a connection has been established, the module transmits packets on this connection. Packets contain sequence numbers as well as the task ID of the corresponding task. The remote master uses the sequence numbers to detect lost packets and sends negative acknowledgments for the missing packets to the mote, which then retransmits the missing packets. End-to-end repair is invoked infrequently because of our hop-by-hop retransmissions (Section 5).

On masters, transport protocols are implemented at user level. Transport and TRD execute in one daemon so that they can share task ID state. On motes, our implementation is engineered to respect mote memory constraints. Retransmission buffers at the sending mote are stored in Flash in order to conserve RAM.[7] Furthermore, our implementation has a configured limit (currently 4) on the number of open connections a mote may have for reliable transport. (Best-effort transport does not have this limit.) All transport protocols work transparently across the two tiers in Tenet. All types of reliable delivery can traverse multiple hops on both tiers of the network and there is almost no functional difference between our implementations for the two tiers.

4.4 Limitations

We view our current work on Tenet as the first step towards a general-purpose architecture for sensor networks. As such, our current prototype lacks functionality required to support some sensor network applications.

Infrastructural components can be easily incorporated into the Tenet system. Our current prototype already includes time synchronization. Once a mature implementation of localization is available, it will be conceptually easy to incorporate this into Tenet. Information from these components (such as location and time) can be exported to applications through tasklets; indeed, we have already developed tasklets that export system information (such as routing state, task concurrency, dynamic memory usage, etc.) and allow applications to read global time and synchronize task execution.

At the moment, the Tenet system does not support delay-tolerant applications, those that require (statistical) delay bounds, or low latency messaging (e.g, Simon et al. [2004]). Whether these can be achieved within the context of this architecture or not remains an open question, one that can only be answered after we have attempted to add support for these in the system. As we state in Section 1, the architectural constraint of Tenet could be relaxed if required, of course, but we aggressively enforce it to most clearly demonstrate the costs and benefits of our approach.

In this article, we have not discussed energy management. Recently, we have been able to retrofit radio duty-cycling into Tenet, without compromising the Tenet prin-

[7]Every new packet sent using stream transport is stored to a circular Flash buffer, which can hold 200 packets in our current implementation. This design trades off energy against RAM. We plan also to explore alternate retransmission strategies.

Task	Max Concurrency	Memory Usage (bytes) Application	Overhead	Data Transmitted (bytes/pkt) Tasking	Output	Overhead
Diagnostics	—	94	22	46	26	46%
Sample[1]→Send	64	64	18	30	6	67%
+ LocalTime	55	76	20	38	14	57%
+ Count	47	90	22	48	20	60%
+ MeanDev	41	104	24	58	26	61%
+ Comparison + DeleteAttr	32	134	28	80	26	61%
Sample[40]→Send	—	142	18	40	84	5%

Fig. 4. Statistics for example tasks. Max Concurrency shows the number of concurrent tasks a single Tmote mode can support. Memory Usage shows the number of bytes used per task for application data and malloc overhead. Data Transmitted shows the number of bytes needed to specify a task in a task dissemination packet, and the number of bytes sent per task execution. The % Overhead column shows how much of that load is taken up by attribute overhead (type and length bytes); when the Sample task is instructed to include 40 samples per attribute instead of one, this overhead drops significantly.

ciple [Gnawali et al. 2009]. Several other extensions to the Tenet system remain open, and we believe these are conceptually easy to achieve. Support for actuation can be naturally expressed as a task. Similarly, mote-tier storage can also be realized by adding appropriate abstractions (reading and writing to named persistent storage) and associated tasklet implementations. Finally, mechanisms for ensuring the authenticity and integrity of data, and methods for multi-user access control and resource management, can borrow from similar mechanisms in other general-purpose systems.

5. TENET EVALUATION

This section evaluates Tenet through microbenchmarks of its tasking and networking subsystems and reliable stream transport, and through four application studies, including a pursuer-evader game (PEG), an application believed to be particularly challenging to implement efficiently without in-network data fusion. We find that Tenet's core mechanisms, such as tasking, scale well; that Tenets are robust and manageable; that its tasking language is flexible enough to accommodate a variety of applications; and that even challenging applications may be implemented with little efficiency loss.

5.1 Tasks and Tasklets

Tenet tasklets and the base Tenet implementation, particularly its tasking and memory subsystems, should be lightweight enough to allow many tasks to execute concurrently on a mote; for instance, to run diagnostics tasks concurrently with sensing tasks. Thus, as an end-to-end evaluation of the Tenet tasking software stack, we find the maximum number of copies of a task that a mote can support. A Tmote can run *32* concurrent versions of a nontrivial task; this maximum concurrency rises for simpler tasks. We also measure other basic aspects of the system, such as memory and packet overhead.

Concurrency. We begin the concurrency study with a simple *sample-and-send* task.

```
Sample(60000ms, 1, REPEAT, ADC0, A) -> Send()
```

We increase the sampling period to 60 seconds to prevent the radio from being the concurrency bottleneck. The fact that the radio cannot support very much traffic, particularly when communicating over multiple hops, is a well-known issue with sensor networks in general. Varying numbers of copies of this task are run alongside the following single *diagnostics* task, which sends, from each mote, a timestamp, the next hop, and statistics of memory usage.

```
Repeat(1000ms) -> LocalTime(A)
    -> MemoryStats(B) -> NextHop(C) -> Send()
```

A single Tmote mote can concurrently execute the *diagnostics* task plus up to 64 *sample-and-send* tasks. Naturally, tasks that contain more tasklets consume more resources, so the mote can execute fewer of them at once. We measure this effect by adding tasklets to the *sample-and-send* task, thus making it more complex. For each incremental addition to the task, we measure how many of that intermediate task can execute on a mote at the same time. The more complicated task is as follows.

```
Sample(20000ms, 1, REPEAT, 1, ADC0, A)
    -> LocalTime(B) -> Count(C, 0, 1) -> MeanDev(D, A)
    -> GT(E, A, 0) -> DeleteAttribute(A) -> Send()
```

This task is similar to the data acquisition and processing example in the previous section, except that we additionally delete the sample for good measure. Figure 4 shows the results: a Tmote can concurrently execute 32 instances of this complicated task.

Memory. We now turn our attention to identifying the resource bottleneck that prevents higher task concurrency. There are two candidates: available MCU cycles and RAM. Radio bandwidth cannot be the bottleneck as our tasks sent sufficiently little data.

Our results indicate that memory is the tasking bottleneck on our motes. Figure 4 shows the total bytes allocated from the heap per task in steady state, and the total number of bytes allocated to manage these heap blocks (two bytes per block). Our Tenet Tmotes have around 5540 bytes available in RAM for the heap and call stack. In the steady state, a *sample-and-send* task uses 82 $(64 + 18)$ bytes allocated from the heap and the *diagnostics* task uses 116 $(94 + 22)$ bytes. Thus, even ignoring the call stack, a Tmote wouldn't be able to support more than $\lfloor (5540 - 116)/82 \rfloor = 66$ concurrently executing *sample-and-send* tasks while a *diagnostics* task is running. In actuality, a Tmote supports 64. On more RAM-constrained motes, this number can be much lower; a Tenet MicaZ, for example, supports only 12 *sample-and-send* tasks.

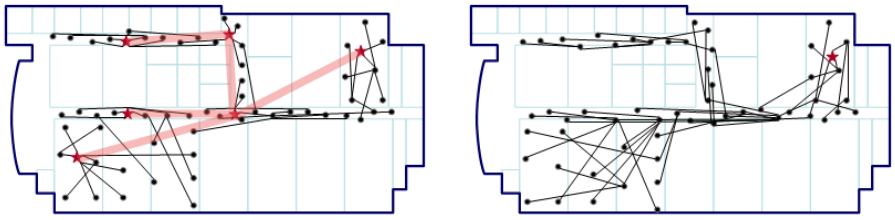
Even tasks that use our most CPU-intensive tasklet, `MeanDev()` (or equivalently `Statistics(MEAN_DEV)`), experience memory constraints before processor constraints. The following task demonstrates this.

```
Sample(1000ms, 1, REPEAT, ADC0, A)
    -> LocalTime(B) -> MeanDev(C, A) -> LocalTime(D)
    -> DeleteAttribute(A) -> Send()
```

This task measures and reports the mean deviation from the mean of a collected sensor value. We record the time before and after running this tasklet to measure

Input Samples	Execution Time (ms)
1	1.8
40	2.3
400	6.8
800	11.9
1200	16.9

Fig. 5. Execution time of measuring the mean deviation from the mean as a function of the number of input samples. Averaged over five runs.



(a) six master topology (b) single master topology

Fig. 6. Testbed topology as gathered by `NextHop()` → `Send()`. Masters are stars, motes are dots. PEG experiments use only the central master.

its execution speed. Our hypothesis is that for the tasklets we have written, none consumes enough MCU cycles such that concurrency will be bounded by the microcontroller. Figure 5 lists the execution times of calculating the mean deviation from the mean as a function of the number of input samples. Even when processing 1200 samples at a time, which consumes about half the application's available RAM, the execution time is a mere 16.9 milliseconds. Thus, a mote running several tasks that gather statistics will only be CPU-bound when the aggregate sampling rate is approximately 71,000 samples per second.

Data Transmitted and Packet Overhead. The task library's processing flexibility results from the use of flexible attribute-based packets and data structures. However, nothing is for free: this flexibility comes at the cost of increased overhead. Figure 4 shows the packet overhead associated with adding type and length fields to each of our data attributes. It requires 30 bytes to specify a task as simple as *sample-and-send*. Each packet generated by this task contains only six bytes of application data, four of which are the name and length of the attribute containing the sample. When sampling is periodic, the ratio of task specification bytes to output bytes becomes insignificant, but for tasks that put a single sample in each packet the overhead of describing the response in TLV format is still large.

To compensate for this overhead, the Sample tasklet can pack more than one sample in an attribute. When the master specifies that 40 samples be packed and sent in each packet (the `Sample[40]` → `Send` task), this overhead drops to 5%.

5.2 Application Case Study: Pursuit-Evasion

How much, if at all, does Tenet degrade application performance relative to a mote-native implementation that performs in-mote multinode fusion? In this section, we examine this question by comparing mote-native and Tenet implementations of a pursuit-evasion application.

Pursuit-Evasion Games. Pursuit-Evasion Games (PEGs) have been explored extensively in robotics research. In a PEG, multiple robots (the pursuers) collectively determine the location of one or more evaders, and try to corral them. The game terminates when every evader has been corralled by one or more robots. PEGs have motivated interesting research directions in multirobot coordination. In this article, however, our interest in PEGs comes from the following observation: in obstructed environments such as buildings, pursuers may not have line-of-sight visibility to evaders, and a sensor network can help detect and track evaders. Indeed, Sharp et al. [2005] describe a mote-level implementation of the mechanisms required for evader detection and tracking in PEGs. In their implementation, the mote network senses evaders using a magnetometer, and transmits a location estimate to one or more pursuers. The network continuously tracks evaders, so that pursuers have an almost up-to-date, if approximate, estimate of where the evaders are. The pursuers can then employ collaborative path planning algorithms to move towards the evaders.

We have reimplemented a version of Sharp et al.'s system, including their *leader election*, *landmark routing*, and *landmark-to-pursuer routing* mechanisms. (We could not use their implementation since it was developed on a previous generation sensor platform, the mica2dot.) Leader election performs in-mote multinode data fusion to determine the centroid of all sensors that detect an evader; the other mechanisms route leader reports to pursuers. Our reimplementation, which we call *mote-PEG*, uses Sharp et al.'s algorithms, but a more mature routing technology, namely MultihopLQI. This allows for a more meaningful comparison with our Tenet implementation.

Tenet and PEGs. PEGs represent a stress test for Tenet. In a dense deployment, it is highly likely that multiple motes will sense the evader. Pushing the application-specific processing into the motes, as mote-PEG does in its leader election code, can conserve energy and reduce congestion. Because Tenet explicitly forbids in-mote multinode fusion, it cannot achieve similar efficiencies.

We have implemented a single pursuer, single evader PEG application for Tenet. In Tenet-PEG, the pursuer is a mobile robot that is part of the master network. The Tenet-PEG application runs on the pursuer, which tasks all the motes to report evader detections whose intensity is above a certain threshold T. The pursuer receives the task responses and computes the evader positions as the centroid of all reports received within a window P, where P is the sampling period at the motes. Extending this implementation to multiple pursuers involves distributing the application across the master tier, which we have left to future work.

Although Tenet-PEG cannot reduce network traffic by multinode data fusion on the motes, it can control overhead by dynamically adjusting the threshold. In our Tenet-PEG implementation, we have implemented a very simple adaptive algorithm. K, the target number of reports, is an input parameter to this algorithm. Initially, our Tenet-PEG implementation sets a low threshold. When it has received at least P (10, in our current implementation) distinct sensor values, it picks the K-th highest sensor value (K is 3 in our experiments), and retasks the motes to report at this threshold. A more sophisticated algorithm would continuously adjust the threshold based on the number of received reports, and is left for future work;

however, even this simple algorithm works well in practice.

Experimental Methodology and Metrics. We now compare the performance of Tenet-PEG and mote-PEG. Our experiments are conducted on the testbed shown in Figure 6(a). This testbed consists of 56 Tmotes and 6 Stargates deployed above the false ceiling of a single floor of a large office building. The Stargate and mote radios are assigned noninterfering channels. The mote radio power is configured such that the maximum network diameter is $5 \sim 7$ hops. This testbed represents a realistic setting for examining network performance as well as for evaluating PEGs. The false ceiling is heavily obstructed, so the wireless communication that we see is representative of harsh environments. The environment is also visually obstructed, and thus resembles, say, a building after a disaster, in which a pursuit-evasion sensor network might aid the robotic search for survivors.

We make two simplifications for our experiments which affect both implementations equally and therefore do not skew the results. First, lacking a magnetometer sensor, we use an "RSSI" sensor. The evader periodically sends radio beacons and sensors detect the existence of the evader by the receipt of the beacons. The beacon's RSSI value is used as an indication of the intensity of the sensed data. Since multiple nodes can detect the evader beacon, its effect is similar to that of having a real magnetometer. The RSSI is also used to implicitly localize the evader; RSSI has been used before for node localization [Bulusu et al. 2000], and since we only require coarse-grained localization (see the following), it is a reasonable choice for our experiments as well. Second, to create a realistic multihop topology, we limit the transmit power of each mote. This results in a topology with a 9-hop diameter, and is comparable to the diameter of the network used in Sharp et al. [2005]. This also results in a realistic tiered network, with the largest distance from a master to a mote being about four hops.

In this setting, since we are interested in how the network affects application performance, we conduct the following experiment. We place one stationary pursuer. An evader tours the floor, carrying a laptop attached to a mote that emits the evader beacon. The frequency of evader beaconing, as well as that of RSSI sampling, is 2 Hz. The laptop receives user input about its current location, and maintains a timestamped log of the evader position. This log represents the ground truth. For Tenet-PEG, we use a network with a single master; this enables a more even comparison with mote-PEG, since using a tiered network could skew performance results in Tenet's favor.

In comparing the two implementations, we use the following metrics. Our first metric measures application-perceived performance, the *error in position estimate*. Many robotic navigation techniques reduce the map of an environment to a topological map [Kuipers and Byun 1988]. This topological map is a collection of nodes and links that represents important positions in the environments. Using such a topological map, path planning can be reduced to graph search [Brooks 1983]. Our Tenet-PEG implementation actually implements a simple graph search technique. In PEG implementations that use such a topological map, the goal is to narrow the evader's location down to the nearest node on the topological map. Thus, our definition of position error at a given instant is the distance on the topological map between the pursuer's estimate of the evader's position, and ground truth. We

	Overhead (msg/min)		
	Min	**Max**	**Average**
Mote-PEG	191	384	272
Tenet-PEG	181	255	217

Fig. 7. PEG application overhead

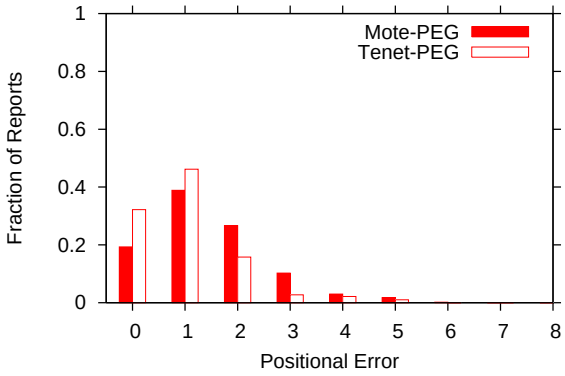


Fig. 8. PEG application error.

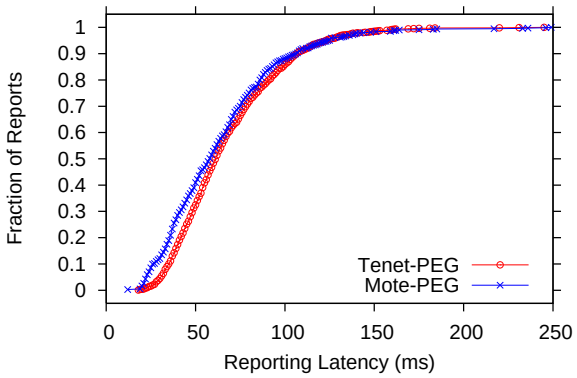


Fig. 9. PEG detection latency.

study the variation of position error over time. In our implementation, we divide our floor into 14 topological nodes, each approximately 22 feet apart.

Our two other metrics measure network performance. The first is the *latency* between when a mote detects an evader and when this detection reaches the pursuer. We measure this using FTSP [Maroti et al. 2004] timestamps. The second is the *application overhead*, the total number of messages received per minute at the pursuer. This measure, while unconventional, indicates the how well the filtering algorithms work. If we get more than the expected 120 reports (at 2 Hz sampling rate) per minute, duplicate information is being received. For Tenet-PEG, this means the RSSI-threshold is too low; for mote-PEG it means that sometimes multiple nodes get elected as leaders.

Results. Figure 8 shows that Tenet-PEG estimates the evader position slightly more accurately than mote-PEG. There are two reasons for this. First, our Tenet-

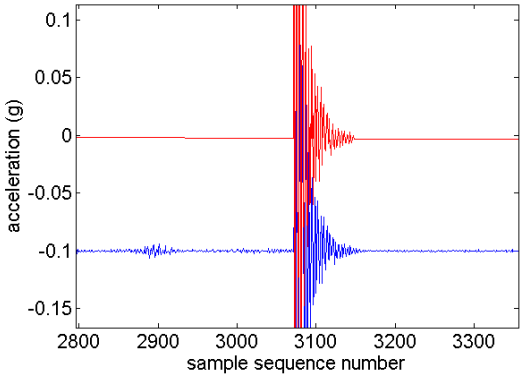


Fig. 10. Vibration sensing with onset detector.

PEG implementation adaptively sets the reporting threshold using information from many nodes across the network. By contrast, mote-PEG's reporting decisions are based on a local leader election, which can sometimes result in spurious local maxima. Second, while mote-PEG computes evader position as the centroid of all the reporting nodes, Tenet-PEG simply uses the position of the reporting node. In mote-PEG, the reporting nodes may sometimes span our floor, resulting in cases where mote-PEG error is several hops.

Figure 9 shows that Tenet-PEG's latency is only marginally higher than that of mote-PEG. This small difference is attributable to the latency across the serial link between the master and its attached mote, as well as processing overheads in the Tenet stack.

Finally, Figure 7 shows that Tenet-PEG incurs slightly lower overhead than mote-PEG. This can also be attributed to Tenet-PEG's adaptive threshold selection algorithm which reduces the number of reporting nodes.

Overall, our results are extremely encouraging. In the case of medium-scale pursuit-evasion, an application previously thought to demand in-mote multinode data fusion for performance, Tenet implementations can perform comparably with mote-native implementations in the experiments conducted on our testbed. Although some applications may use in-mote multinode data fusion, our Tenet-PEG experience suggests that it might be possible to achieve similar performance gains by careful local processing.

5.3 Application Case Study: Event-Based Vibration Monitoring

In Section 5.1, we show several examples of debugging and maintenance tasks that are trivial to express in Tenet. It is difficult to measure quantitatively whether Tenet simplifies the programming of more realistic applications. In this, and subsequent sections, we discuss a few application case studies in which we have implemented realistic (and qualitatively different) applications using Tenet. Two of these applications have been deployed for up to several days in realistic environments. Our first example is a Tenet implementation of a mature structural monitoring system, Wisden [Paek et al. 2005; Xu et al. 2004], an event-based vibration data acquisition system that reliably transmits vibration samples from a network of motes to a base

station. Each mote transmits only interesting events using a simple *onset detector* [Paek et al. 2005], greatly reducing communication requirements. An onset is defined to be a part of a signal waveform that exceeds the mean amplitude by more than a few standard deviations. Wisden implements specialized mechanisms for time synchronization, as well as for reliably transmitting data to a base station.

To port Wisden to Tenet, we implemented an `OnsetDetector` tasklet and used it to build a task *functionally equivalent* to Wisden's mote code. Specifically, the application tasks motes using a task description of the following form.

```
SampleMDA400(20ms, 40, CONT, Z_AXIS, A) -> OnsetDetector(A) -> SendStr()
```

Here, `SampleMDA400()` controls a vibration sensorboard and `SendStr()` invokes our stream transport protocol.

Figure 10 shows the vibration time-series on two MicaZ motes using a single master. One mote was programmed with the preceding task, but *without* `OnsetDetector()` (the lower timeseries). The other was programmed with the preceding task (the upper timeseries). We then put the two motes on a table, and hit the table. The mote without `OnsetDetector()` sends vibration data even before the onset; this vibration is induced by human activity (movement, typing). The other task does not send spurious vibrations, resulting, in our small experiment, in a 90% traffic reduction.

This Tenet implementation of Wisden is not only functionally equivalent to Wisden but improves over Wisden in several aspects. First, a Tenet deployment is more scalable since the bandwidth capacity limit which bounded the number of nodes in Wisden is circumvented in Tenet by using masters. Second, the Tenet implementation of Wisden can alter application parameters (such as sampling frequency, channels, etc.) at runtime by retasking the motes whenever the application wishes to; by contrast, Wisden required reprogramming of the motes. Finally, in Tenet, other application tasks can run concurrently while running the Wisden application. This not only promotes reusability within the network, but it can also provide useful information about the state of network (routing topology, time-sync state, and memory usage) while an application executes, thereby improving the manageability and the robustness of the sensor network.

The Tenet version of Wisden's mote code reduces in the end to three simple tasklets. Tenet integrates support for tasking and stream transport, making the Wisden application itself both smaller and simpler. Our qualitative judgment is that, even for these reasons alone, Tenet significantly simplifies application development.

5.4 Real-World Deployment: Vibration Monitoring at Vincent Thomas Bridge

In this section, we discuss our experiences from a real-world deployment of Tenet on a large suspension bridge. This Tenet deployment ran an application designed to continuously monitor the bridge's vibrations, and reliably transmit the data back to a base station.

Deployment Methodology and Experiences. We deployed a Tenet system, running a simple structural data acquisition application, on Vincent Thomas Bridge, a 6,000 ft long suspension bridge with a main suspension span of 1,500 ft, and a height of

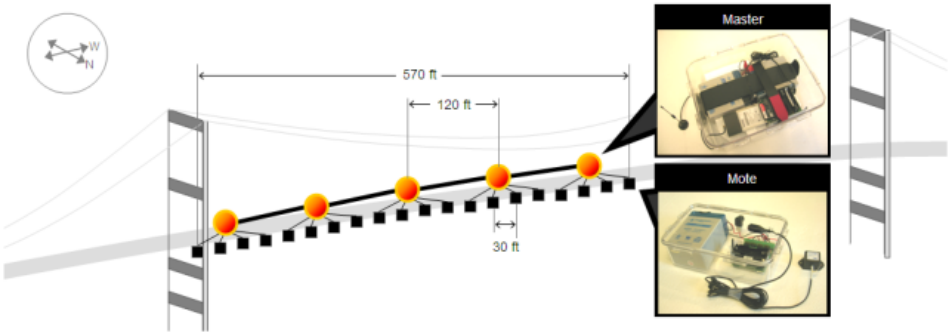


Fig. 11. Deployment topology on the bridge.

185 ft above water. We instrumented 600 ft of the bridge using a Tenet network consisting of 5 master nodes and 20 motes, and collected data for 24 hours. Masters and motes were equipped with batteries capable of lasting the entire duration of the experiment.

The communication environment on the bridge was relatively harsh. Masters were able to communicate at distances of only 100 ft with less than 1% packet loss rate. Beyond 140 ft, they were unable to communicate. Motes were able to communicate only 80 ft reliably, and there was a sharp drop off of connectivity at 90 ft. These results were surprisingly worse than what we could achieve in a ground-level open space, and dictated our deployment topology.

The deployment spanned 600 ft starting from the east tower towards the center of the bridge (Figure 11). The network was configured as a linear topology of 20 motes and 5 masters. Each mote was placed 30 ft apart from each other resulting in a maximum distance of 45 ft from any mote to its nearest master. Every pair of neighboring masters was separated by about 120 ft.

Vibration Monitoring Application. On this network, we ran a Tenet application which tasks all the motes to sample at 20 Hz along three axes (x,y,z), retrieves the data, and stores the responses into a log file at a master. Although our hardware and the Tenet system permit a sampling frequency of up to 1 kHz, it is not possible to stream data continuously at such a high rate given the capacity limits on the motes. Based on prior experiences with hardware limitations [Paek et al. 2005], we concluded that our system would be able to support continuous-sampling and real-time transmission without any compression at 80 Hz for a single channel, or one-third of that for three channels. To be conservative, we selected 20 Hz tri-axis for our experiment, and used the following Tenet task.

```
SampleMda400(50ms, 20, CONT, AX, A, AY, B, AZ, C) -> SendStr()
```

This task is similar to that of the Wisden application (Section 5.3) except that here we do not perform `OnsetDetector()` processing. Recall that `SendStr()` invokes reliable stream transport protocol. Hence every node continuously generates 3 packets per second (where each packet contains 20 samples and each sample is 16 bits), which adds up to total of 60 packets per second being received at the application. This corresponds to a data rate of 35.52 kbps (including a timestamp

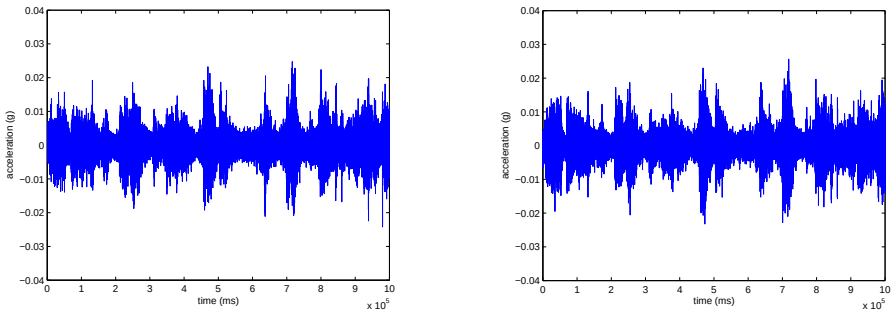


Fig. 12. Time-series plot of the acceleration data from two different nodes: measured in perpendicular direction to the bridge, at around 5:00pm.

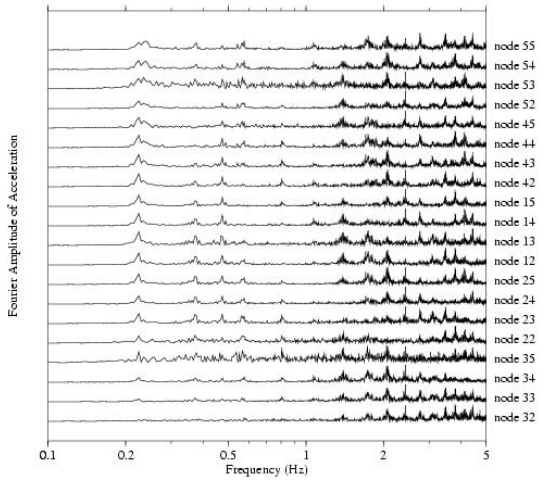


Fig. 13. Vertical vibration modal frequencies (Hz).

and packet headers), a challenging rate for a flat multihop network of 20 motes especially when reliable data transfer is a requirement [Kim et al. 2007; Paek and Govindan 2007].

Data Validation. Figure 12 shows that the time-series plot of the ambient vibration data from two different nodes on the bridge are consistent with each other. Figure 13 depicts the spectral density plots for data from several sensors, taken at about the same time. FFT profiles of the data have all been bandpass filtered for frequencies between 0.15 and 5.0 Hz, using a two-pass, two-pole Butterworth filter. The spectral density plots show two interesting features. First, they are internally consistent: many peaks (*modes*) are observed at several sensors. That some modes are not visible at some sensors is expected; such sensors are placed at nulls of the corresponding vibration mode shapes. Second, the location of the modes is also consistent with published results obtained from wired instrumentation [Smyth et al. 2003].

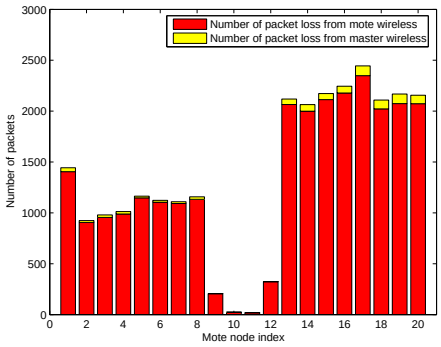


Fig. 14. Number of packet retransmissions due to master links and mote links

System Evaluation. Our experiment ran for 26 hours, comprising 22 *rounds*: in each round the system collects data for 60 minutes, backs up the data, and resets the motes during the next 10 minutes. This experiment design was an artifact of the fact that we had observed rare software crashes on the motes. Rebooting the entire network every hour enabled us to recover from these crashes reliably. Out of 22 rounds, 19 rounds had data from all 20 nodes, and the other 3 rounds had data from 19 nodes, which results in a node-round yield of 99.32% (437 out of 440 node-rounds). In each of those three incomplete rounds, one node was not able to start the task as a result of the software bug.

We collected a total of 6,430,792 data packets during the experiment which amounts to approximately 323,745 packets from each sensor node. Each of the 19 complete datasets have approximately 294,500 packets and 4,386,000 samples per round.

For all 437 node-rounds in which we were able to collect data, our system achieved 100% end-to-end reliability with no lost packets. To achieve this, the end-to-end reliable transport protocol was required to retransmit 26,965 packets, which is 0.42% of the total number of data packets received (Figure 14). Nodes farther away from center of the network required more retransmissions because their packets traversed longer paths (more hops) to get to the collection point. Among these retransmissions, approximately 2.92% were due to packet loss on the IEEE 802.11b master wireless links and 97.08% were due to loss on the IEEE 802.15.4 mote wireless links (Figure 14). Despite losses on the mote as well as master wireless links, our system was able to reliably collect all the transmitted samples.

Overall, the results show that Tenet performs well in real-world deployment. Its tiered architecture scales network capacity and allows reliable delivery of high rate data that would otherwise have been difficult to achieve on a flat multihop network of motes.

5.5 Real World Deployment: Pitfall Trap Monitoring at James Reserve

In this section, we discuss a qualitatively different real-world deployment of Tenet: an imaging application for pitfall trap monitoring, deployed at the James Reserve[8].

[8] `http://www.jamesreserve.edu/+`

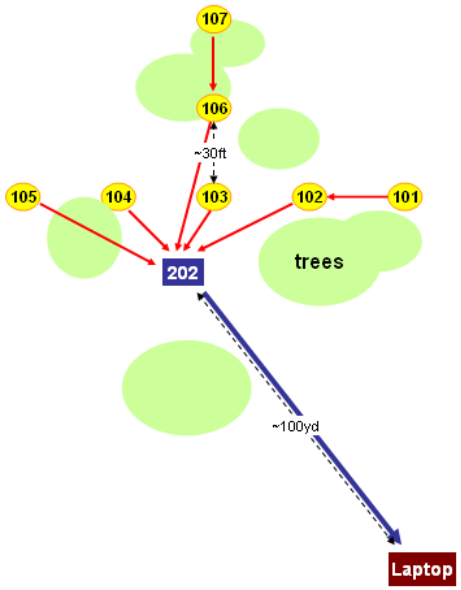


Fig. 15. Deployment topology at the James Reserve.

This application uses image-sensors (*Cyclops* cameras [Rahimi et al. 2005]) and leverages the Cyclops' on-board processing capabilities. It is also the most complicated Tenet application we have explored to date, and demonstrates the expressiveness and reusability of Tenet's tasking library.

Pitfall trap arrays at James Reserve are used by biologists to sample the local population of lizards and amphibians. Biologists deploy arrays of traps in clusters. When a lizard is caught in a trap, it is tagged and freed. Over time, a count of trapped lizards can be used to estimate the overall population of lizards. However, if a lizard is left too long in a trap, it can die. So, biologists frequently poll each trap by physically visiting each every few hours: because these traps are deployed over a few square kilometers, inspecting these traps can be an arduous task. A Tenet network with image sensors can help: the biologists can inspect the images remotely, and only visit a trap when a lizard is caught, saving time, effort, and frustration.

We deployed a Tenet system in an array of pitfall traps at James Reserve. Our deployment consisted of seven traps in a star configuration. We placed 7 traps each equipped with a mote and attached Cyclops camera, one stargate master near the array, and a laptop master at a lodge about 100 yds away from the array. Figure 15 depicts the deployment topology. Some of the nodes were behind the trees with foliage which blocked line of sight to the base station.

Our goal was to design a triggered data collection system, where each node frequently captures an image, but only sends it to the base station when some interesting change has been detected in the image. This is necessary to conserve network bandwidth, since each image is 16 KB. However, the image change detection algorithm we use can exhibit false negatives. If a false negative detection occurs, no image will be delivered at that instant. Even if there actually was a lizard in that

trap, if that lizard does not move, no image transfer will be triggered at subsequent sampling times as well. This might result in a missed and eventually dead lizard. To avoid this, we designed our application to transfer at least one image every 30 minutes. Here is a Tenet task that realizes this relatively complicated logic.

```
Repeat(120000ms)
    -> ImageDetect(TAKE_NEW, FLASH_ON, 128x128, BW, A)
    -> Count(B,0,1) -> Mod(B,B,'15') -> Eq(B,B,'0')
    -> Or(A,A,B) -> Store(C,A) -> DeleteAllAttributeIf(A)
    -> Send(E2E_ACK)
    -> Retrieve(C) -> Not(C,C) -> DeleteActiveTaskIf(C)
    -> DeleteAttribute(C) -> ImageDetect(RESET)
    -> ImageFetch(LAST, 40, 140ms, D)
    -> SendStr();
```

In this task, `ImageDetect()` invokes a background-subtraction-based algorithm to detect noticeable differences between the new and previous image, and `ImageFetch()` retrieves the last image stored in the Cyclops memory. Thus the preceding task takes an image every 2 minutes, and transfers it only if some change has been detected in the image or every 15th run (every 30 minutes). If an object is *not* detected and it is *not* the 15th run, the task sends a small message ($A \equiv 0$) using `Send(E2E_ACK)`, as a keep-alive. Otherwise, it transfers the last image using `SendStr()`, and resets the detection background. Each image is an 128x128 black-and-white image whose size is 16KB, and each packet can contain up to 40 bytes of image fragment data; so, each image requires 410 packets.

These `Image`-style tasklets are different from other tasklets in that they are executed on the cameras themselves. Image processing requires memory and MCU cycles beyond the capabilities of the current generation of motes, and the Cyclops board was designed to provide specialized image processing tasks. For this reason, the Tenet mote exports single `Image()` tasklet that can be used to parametrize and invoke `Image`-related operations, such as `ImageDetect(), ImageFetch(), ImageSetParam(),` etc. which are implemented within the camera themselves. This design allows any resource-intensive operation to be performed off-mote and can be applied to other specialized external sensors (e.g., high-frequency ADC boards that can sample at 10 kHz) with on-board processing.

Our deployment lasted 3 days, and the network was operational only during the daytime hours (7am–7pm). We collected 589 Cyclops images, out of which 588 images were complete; there was one incomplete image from a node which almost ran out of battery (we used two D-cell batteries on each node). Figure 16 plots the time when each has transferred an image on the last day of the experiment. Many of the transferred images were triggered as a result of changes in the intensity of sunlight (Figure 18), and some others were intentionally triggered by us to test the system. One set of images, at node 102 between 7:45pm and 8:40pm, caught a trapped spider (Figure 17).

Finally, note that there was a similar deployment at James Reserve before [Ahmadian et al. 2008] that did not use Tenet. Our deployment not only replaced the previous system but also improved upon that in several aspects: multihop communication, reliable data delivery, easier-to-use end-to-end system, etc. This is another example which shows that Tenet is reusable and expressive: Tenet can be

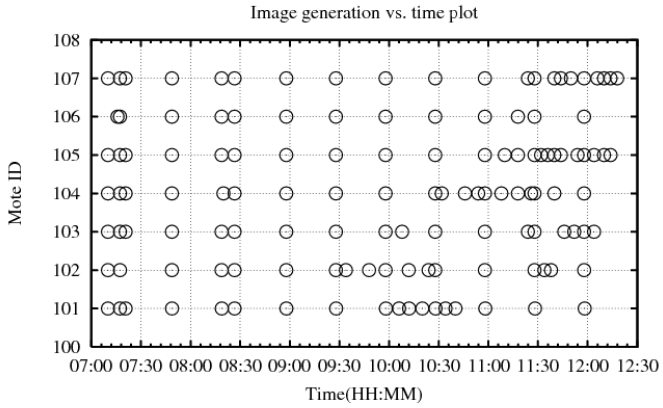


Fig. 16. Image-transfer vs. time plot: An image is generated every 30 minutes, in addition to whenever an object is detected.

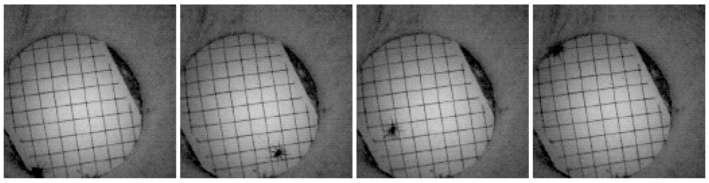


Fig. 17. Spider triggered image transfers.

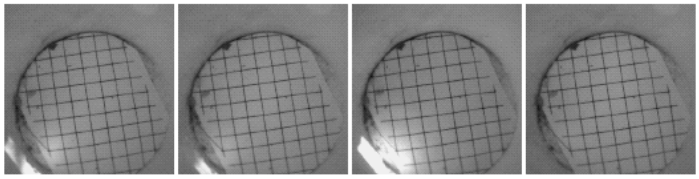


Fig. 18. Sun/shade change caused image transfers.

used to fully implement and improve on previously existing applications.

In summary, this application has shown the expressiveness and reusability of the Tenet tasking library, and how off-board processing can be expressed in it. Largely based on experiences from this deployment, we have moved on to a larger Tenet deployment with 4 masters and 20 cameras for bird nest monitoring at James Reserve [Hicks et al. 2008]. This deployment incorporates image compression, and uses a recently design congestion control protocol, RCRT [Paek and Govindan 2007].

5.6 Manageability

Network monitoring is an important part of networked system manageability. Although we have not extensively evaluated manageability, we have been able to quite easily construct simple applications that can monitor and measure a Tenet network. For example, the following task allows us to get a snapshot of the routing tree at

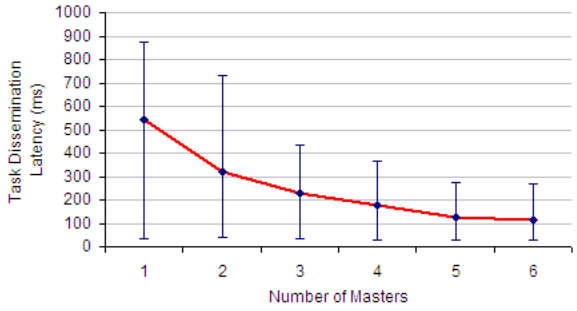


Fig. 19. Task dissemination latency.

any instant.

```
Wait(1000ms) -> NextHop(A) -> Send(E2E_ACK)
```

This obtains each mote's routing parent using our reliable packet delivery protocol. Figure 6 was gathered in this manner. Such an application can be invaluable for monitoring and debugging, particularly since it can run concurrently on Tenet with an application which we may be trying to debug.

It is also easy to perform certain kinds of measurements. This task

```
GlobalTime(A) -> Wait(1000ms) -> Send(E2E_ACK)
```

timestamps a task using the time computed by the FTSP [Maroti et al. 2004] protocol (which is integrated into the stack) as soon as it is received, backs off for a second to reduce congestion, and sends the results back. It can be used to measure the latency of task dissemination. We have run this on the network shown in Figure 6 and measured the Tenet's task dissemination latency with a varying number of masters turned on. With 6 masters, the average tasking latency is 110 ms, and the largest is 550 ms (Figure 19). The benefits of tiering are clearly evident; in our testbed, the average tasking latency using 6 masters is about a fifth of that using only 1 master (because the network has a much smaller mote diameter, leaving fewer opportunities for lost packets and retransmissions between motes).

5.7 Robustness

We conducted a simple experiment to demonstrate the robustness of our current Tenet implementation to the failure of masters. In this experiment, we tasked 35 nodes in a 5-master network to sample their temperature sensor and transmit the samples using our reliable transport protocol. The task chain for this experiment was as follows.

```
Sample(100ms, 1, REPEAT, ADC10, A) -> SendStr()
```

Five minutes after tasking the network, we turned off one of the masters, and five minutes thereafter, another.

Figure 20 plots, for each connection, the received sequence number against the time at which the corresponding packet was received at the master. Two things are noticeable about the figure. For all connections, initially, the sequence number

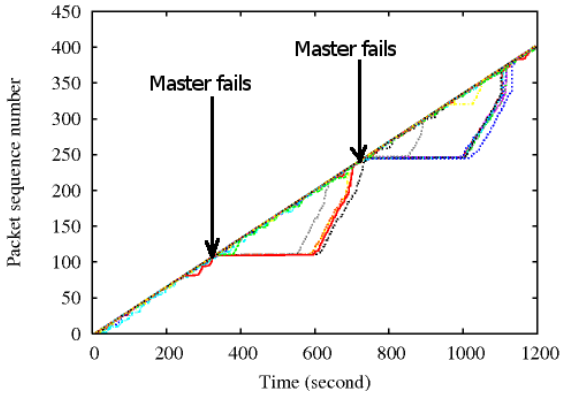


Fig. 20. Sequence number evolution of stream transport connections.

evolves smoothly, showing relatively few packet losses. When the first master is turned off, the sequence number evolution of some (but not all) of the motes exhibits a discontinuity; until routing converges again, many packets are lost. Eventually, however, those connections do recover and packets from the affected nodes are retransmitted to the master. We attribute the long routing convergence times to the high traffic rate in the network. A similar behavior is seen when the second master is turned off.

We also measured the rate of negative acknowledgements to estimate the efficacy of stream transport. For sources that were not affected by the route change, only 7% of the packets were lost end-to-end, despite a fairly heavy traffic load. By constrast, the number was 25% for those sources that were affected by master failure. Finally, using our tasking latency measurement tool, we measured the latency before and during this experiment; the average tasking latency increased threefold (from 137 ms to 301 ms).

6. CONCLUSIONS

In this article, we have shown that the Tenet architecture simplifies application development for tiered sensor networks without significantly sacrificing performance. By constraining multi-node fusion to the master tier, Tenet also benefits from having a generic mote-tier that does not need to be customized for applications. Our Tenet system is able to run applications concurrently, and our collections of tasklets support data acquisition, processing, monitoring, and measurement functionality. Many interesting research directions remain: energy management, support for mobile elements, network congestion control, extending the task library to incorporate a richer tasklet set, and so forth.

APPENDIX

A.1 List of Tasklet API's

This section contains the list of tasklet APIs that are provided by the tasklets in Figure 2. Unless otherwise stated, the default data type returned by a tasklet is a vector of unsigned 16-bit integer(s). All operations (e.g., Arith, Logical, etc.)

take attribute(s) and/or constants as their arguments. Unless otherwise stated, all operations on attribute(s) assume that an attribute is a vector of unsigned 16-bit integers, and that two arguments of an operation are either of equal size or a scalar (vector of size one or a constant). Tasklet APIs are case-insensitive.

Tasking API	**Description**	**Tasklet**
Count	return a value which is incremented everytime it runs.	Count
Constant	return a constant	Count
Issue	issue a task after delay (global or relative, once or periodic)	Issue
Wait	delay a task for 'period'	Issue
Repeat	repeat a task periodically every 'period'	Issue
Alarm	run the task at global time 'starttime'	Issue
GlobalRepeat	repeat a task every 'period' after global 'starttime'	Issue
Get	return a system information	Get
NextHop	return routing next-hop (parent)	Get
GlobalTime	return 32-bit time-synchronized global time	Get
LocalTime	return 32-bit local time	Get
RfPower	return RF power configuration	Get
RfChannel	return RF channel configuration	Get
Memory_Stats	return dynamic memory stats structure	Get
Leds	return the state of the LEDs	Get
Num_Tasks	return the number of tasks installed	Get
Num_Active_Tasks	return the number of active tasks running	Get
is_Timesync	return whether time is synchronized or not	Get
Local_Address	return node id	Get
Platform	return platform type	Get
Clock_Freq	return clock frequency used for localtime	Get
Master	return current nearest routing master	Get
HopCount	return hop count to nearest routing master	Get
Rssi	return RSSI value from routing next-hop	Get
Logical	Perform the logical operation on attribute(s)	Logical
And	$result \leftarrow attr \ \& \ arg$	Logical
Or	$result \leftarrow attr \mid arg$	Logical
Not	$result \leftarrow \overline{arg}$	Logical
Bit	Perform the bit operation on attribute(s)	Bit
Bit_And	$result \leftarrow attr \ AND \ arg$	Bit
Bit_Or	$result \leftarrow attr \ OR \ arg$	Bit
Bit_Not	$result \leftarrow NOT \ arg$	Bit
Bit_Xor	$result \leftarrow attr \ XORarg$	Bit
Bit_Nand	$result \leftarrow attr \ NAND \ arg$	Bit
Bit_Nor	$result \leftarrow attr \ NOR \ arg$	Bit
ShiftLeft	$result \leftarrow attr \ll arg$	Bit
ShiftRight	$result \leftarrow attr \gg arg$	Bit
Arith	perform arithmetic operation	Arith
Add	$result \leftarrow attr + arg$	Arith
Sub	$result \leftarrow attr - arg$	Arith
Mult	$result \leftarrow attr \times arg$	Arith
Div	$result \leftarrow attr \div arg$	Arith
Diff	$result \leftarrow \lvert attr - arg \rvert$	Arith
Mod	$result \leftarrow attr \ \% \ arg$	Arith
Pow	$result \leftarrow attr^{arg}$	Arith
Comparison	perform comparison operation	Comparison
LT	$result \leftarrow (attr < arg \ ?)$	Comparison
GT	$result \leftarrow (attr > arg \ ?)$	Comparison
EQ	$result \leftarrow (attr \equiv arg \ ?)$	Comparison
LEQ	$result \leftarrow (attr \leq arg \ ?)$	Comparison
GEQ	$result \leftarrow (attr \geq arg \ ?)$	Comparison
NEQ	$result \leftarrow (attr \neq arg \ ?)$	Comparison
Count_LT	$result \leftarrow count(attr < arg \ ?)$	Comparison
Count_GT	$result \leftarrow count(attr > arg \ ?)$	Comparison
Count_EQ	$result \leftarrow count(attr \equiv arg \ ?)$	Comparison
Count_LEQ	$result \leftarrow count(attr \leq arg \ ?)$	Comparison
Count_GEQ	$result \leftarrow count(attr \geq arg \ ?)$	Comparison
Count_NEQ	$result \leftarrow count(attr \neq arg \ ?)$	Comparison

Stats	perform statistical operation	Stats
Sum	$result \leftarrow sum(attr)$	Stats
Min	$result \leftarrow minimum(attr)$	Stats
Max	$result \leftarrow maximum(attr)$	Stats
Avg	$result \leftarrow average(attr)$	Stats
Cnt	$result \leftarrow count(attr)$	Stats
MeanDev	$result \leftarrow mean_deviation(attr)$	Stats
Attribute	check an attribute in the current active task	Attribute
Exist	return 1 if attr exists, otherwise 0	Attribute
Not_Exist	return 1 if attr does not exist, otherwise 0	Attribute
Length	return the length of attr vector	Attribute
Actuate	actuates a particular channel	Actuate
Set_Leds	set the leds to according to given value	Actuate
Sounder	start/stop Micasb sounder	Actuate
Storage	storage is valid within a task, maintained across active tasks	Storage
Store	store an attribute into storage	Storage
Retrieve	retrieve an attribute from storage	Storage
Pack	pack consecutive input values into a vector of values	Pack
Pack_n_Wait	perform 'pack' and wait until the vector is full	Pack
Send	send task response back using best-effort transport	SendPkt/Str/Rcrt
SendPkt	send task response back using packet transport	SendPkt
SendStr	send task response back using stream transport	SendStr
SendRcrt	send task response back using RCRT protocol	SendRcrt
Reboot	reboot the mote	Reboot
DeleteAttributeIf	delete an attribute if arg == TRUE	DeleteAttributeIf
DeleteAttribute	delete an attribute	DeleteAttributeIf
DeleteAllattributeIf	delete all attributes if arg == TRUE	DeleteAttributeIf
DeleteTaskIf	delete the task completely if arg == TRUE	DeleteTaskIf
DeleteActiveTaskIf	delete active instance of the task if arg == TRUE	DeleteActiveTaskIf
Sample	sample on-board ADC's	Sample
Voltage	return voltage level	Voltage
UserButton	(Telosb only) run the task everytime user button is pressed	UserButton
SampleMDA400	(Micaz/Mica2 only) sample mda400 vibration board	SampleMda400
Onset_Detector	perform onset-detection and data-filtering	OnsetDetector
Image	(Micaz/Mica2 only) interact with Cyclops camera	Image
Image_Snap	take a picture	Image
Image_Detect	perform background-subtraction based object detection	Image
Image_Get	get an image from Cyclops	Image
Image_Set_Capture_Params	sets camera parameters	Image
Image_Get_Capture_Params	gets camera parameters	Image

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Feng Zhao for their comments. We also thank David Culler, Joseph Polastre, and the other attendees of the first SNA workshop in Berkeley for inspiring discussions on Tenet and for suggesting PEG as a test case. We thank Monica Kohler for advice on processing data from the bridge deployment, and other CENS researchers for collaboration in the deployment at James Reserve.

REFERENCES

Aguayo, D., Bicket, J., Biswas, S., Couto, D. D., and Morris, R. The mit roofnet project. `http://pdos.csail.mit.edu/roofnet/`.

Ahmadian, S., Ko, T., Hicks, J., Rahimi, M., Estrin, D., Soatto, S., and Coe, S. 2008. Heartbeat of a nest: using imagers as biological sensors. *In submission to ACM Transactions on Computational Logic*.

Arora, A., Ramnath, R., Ertin, E., Sinha, P., Bapat, S., Naik, V., Kulathumani, V., Zhang, H., Cao, H., Sridharan, M., Kumar, S., Seddon, N., Anderson, C., Herman, T., Trivedi, N., Zhang, C., Nesterenko, M., Shah, R., Kulkarni, S., Aramugam, M., Wang, L., Gouda, M., Choi, Y.-r., Culler, D., Dutta, P., Sharp, C., Tolle, G., Grimmer, M., Ferriera, B., and Parker, K. 2005. *ExScal*: Elements of an extreme scale wireless sensor network. In

Proceedings of 11th IEEE International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA '05).

Brooks, R. A. 1983. Solving the find-path problem by good representation of free space. *IEEE Transactions on Systems, Man and Cybernetics 13,* 3 (Mar./Apr.), 190–197.

Bulusu, N., Heidemann, J., and Estrin, D. 2000. GPS-less low cost outdoor localization for very small devices. *IEEE Personal Communications 7,* 5 (Oct.), 28–34.

Chu, D., Popa, L., Tavakoli, A., Hellerstein, J. M., Levis, P., Shenker, S., and Stoica, I. 2007. The design and implementation of a declarative sensor network system. In *Proceedings of 5th ACM International Conference on Embedded Networked Sensor Systems (SenSys'07).*

Clausen, T. and Jacquet, P. 2003. Optimized link state routing protocol (olsr).

Culler, D., Dutta, P., Ee, C. T., Fonseca, R., Hui, J., Levis, P., Polastre, J., Shenker, S., Stoica, I., Tolle, G., and Zhao, J. 2005. Towards a sensor network architecture: Lowering the waistline. In *Proceedings of 10th Hot Topics in Operating Systems Symposium (HotOS-X).* 139–144.

Dunkels, A., Österlind, F., and He, Z. 2007. An adaptive communication architecture for wireless sensor networks. In *Proceedings of 5th ACM International Conference on Embedded Networked Sensor Systems (SenSys'07).*

Estrin, D., Govindan, R., Heidemann, J., and Kumar, S. 1999. Next century challenges: Scalable coordination in sensor networks. In *Proceedings of 5th Annual International Conference on Mobile Computing and Networking (MobiCom'99).* 263–270.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. 2003. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI).* 1–11.

Gnawali, O., Na, J., and Govindan, R. 2009. Application-informed radio duty-cycling in a re-taskable multi-user sensing system. In *Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'09).*

Greenstein, B., Kohler, E., and Estrin, D. 2004. A sensor network application construction kit (SNACK). In *Proceedings of 2nd ACM International Conference on Embedded Networked Sensor Systems (SenSys'04).* 69–80.

Greenstein, B., Mar, C., Pesterev, A., Farshchi, S., Kohler, E., Judy, J., and Estrin, D. 2006. Capturing high-frequency phenomena using a bandwidth-limited sensor network. In *Proceedings of 4th ACM International Conference on Embedded Networked Sensor Systems (SenSys'06).*

Gupta, P.; Kumar, P. 2000. The capacity of wireless networks. *IEEE Transactions on Information Theory 46,* 2 (Mar.), 388–404.

Guy, R., Greenstein, B., Hicks, J., Kapur, R., Ramanathan, N., Schoellhammer, T., Stathopoulos, T., Weeks, K., Chang, K., Girod, L., and Estrin, D. 2006. Experiences with the Extensible Sensing System ESS. Tech. Rep. 61, CENS. Mar. 29.

Han, C.-C., Kumar, R., Shea, R., Kohler, E., and Srivastava, M. 2005. A dynamic operating system for sensor nodes. In *Proceedings of 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys'05).*

He, T., Stankovic, J., Stoleru, R., Gu, Y., and Wu, Y. 2008. Essentia: Architecting wireless sensor networks asymmetrically. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE.*

Hicks, J., Paek, J., Coe, S., Govindan, R., and Estrin, D. 2008. An easily deployable wireless imaging system. In *Proceedings of ImageSense08: Workshop on Applications, Systems, and Algorithms for Image Sensing.*

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. 2000. System architecture directions for network sensors. In *Proceedings of 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX).* 93–104.

Hnat, T. W., Sookoor, T. I., Hooimeijer, P., Weimer, W., and Whitehouse, K. 2008. Macrolab: a vector-based macroprogramming framework for cyber-physical systems. In *Proceedings of 6th ACM International Conference on Embedded Networked Sensor Systems (SenSys'08)*.

Hui, J. and Culler, D. 2004. The dynamic behavior of a data dissemination algorithm at scale. In *Proceedings of 2nd ACM International Conference on Embedded Networked Sensor Systems (SenSys'04)*.

Intanagonwiwat, C., Govindan, R., and Estrin, D. 2000. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of 6th Annual International Conference on Mobile Computing and Networking (MobiCom'00)*. 56–67.

Kim, S., Fonseca, R., Dutta, P., Tavakoli, A., Culler, D., Levis, P., Shenker, S., and Stoica, I. 2007. Flush: A reliable bulk transport protocol for multihop wireless networks. In *Proceedings of 5th ACM International Conference on Embedded Networked Sensor Systems (SenSys'07)*. 351–365.

Kothari, N., Gummadi, R., Millstein, T., and Govindan, R. 2007. Reliable and efficient programming abstractions for wireless sensor networks. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*.

Kottapalli, V. A., Kiremidjian, A. S., Lynch, J. P., Carryer, E., Kenny, T. W., Law, K. H., and Lei, Y. 2003. Two-tiered wireless sensor network architecture for structural health monitoring. In *Proceedings of SPIE 5057—Smart Structures and Materials 2003: Smart Systems and Nondestructive Evaluation for Civil Infrastructures*, S.-C. Liu, Ed.

Kuipers, B. J. and Byun, Y.-T. 1988. A robust qualitative method for spatial learning in unknown environments. In *Proceedings of 7th National Conference on Artificial Intelligence (AAAI-88)*.

Levis, P. and Culler, D. 2002. Maté: A tiny virtual machine for sensor networks. In *Proceedings of 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*. 85–95.

Levis, P., Gay, D., and Culler, D. 2005. Active sensor networks. In *Proceedings of 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'05)*. USENIX.

Levis, P., Patel, N., Culler, D., and Shenker, S. 2004. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI'04)*. USENIX.

Liu, B., Liu, Z., and Towsley, D. 2003. On the capacity of hybrid wireless networks. In *Proceedings of 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'03)*.

Liu, J., Cheong, E., and Zhao, F. 2005. Semantics-based optimization across uncoordinated tasks in networked embedded systems. In *Proceedings of 5th ACM Conference on Embedded Software (EMSOFT 2005)*.

Liu, J. and Zhao, F. 2005. Towards semantic services for sensor-rich information systems. In *Proceedings of 2nd International Conference on Broadband Networks (BROADNETS 2005)*. 44–51.

Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. 2002. TAG: A Tiny AGgregation service for ad-hoc sensor networks. In *Proceedings of 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*.

Maroti, M., Kusy, B., Simon, G., and Ledeczi, A. 2004. The flooding time synchronization protocol. In *Proceedings of 2nd ACM International Conference on Embedded Networked Sensor Systems (SenSys'04)*.

Mhatre, V. P., Rosenberg, C., Kofman, D., Mazumdar, R., and Shroff, N. 2005. A minimum cost heterogeneous sensor network with a lifetime constraint. *IEEE Transactions on Mobile Computing 4,* 1 (Jan./Feb.), 4–15.

Newton, R., Morrisett, G., and Welsh, M. 2007. The regiment macroprogramming system. In *Proceedings of 6th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'07)*.

Paek, J., Chintalapudi, K., Cafferey, J., Govindan, R., and Masri, S. 2005. A wireless sensor network for structural health monitoring: Performance and experience. In *Proceedings of The Second IEEE Workshop on Embedded Networked Sensors, (EmNetS-II)*. 1–10.

Paek, J. and Govindan, R. 2007. RCRT: Rate-Controlled Reliable Transport for Wireless Sensor Networks. In *Proceedings of 5th ACM International Conference on Embedded Networked Sensor Systems (SenSys'07)*.

Polastre, J., Hui, J., Levis, P., Zhao, J., Culler, D., Shenker, S., and Stoica, I. 2005. A unifying link abstraction for wireless sensor networks. In *Proceedings of 3th ACM International Conference on Embedded Networked Sensor Systems (SenSys'05)*. 76–89.

Polastre, J., Szewczyk, R., and Culler, D. 2005. Telos: Enabling ultra-low power wireless research. In *Proceedings of 4th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'05), SPOTS track*.

Rahimi, M., Baer, R., Iroezi, O. I., Garcia, J. C., Warrior, J., Estrin, D., and Srivastava, M. 2005. Cyclops: In situ image sensing and interpretation in wireless sensor networks. In *Proceedings of 3th ACM International Conference on Embedded Networked Sensor Systems (SenSys'05)*. 192–204.

Ratnasamy, S., Karp, B., Yin, L., Yu, F., Estrin, D., Govindan, R., and Shenker, S. 2002. GHT: A geographic hash table for data-centric storage. In *Proceedings of 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*. ACM, 78–87.

Rhee, S., Seetharam, D., and Liu, S. 2004. Techniques for minimizing power consumption in low data-rate wireless sensor networks. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC 2004)*.

Saltzer, J. H., Reed, D. P., and Clark, D. D. 1984. End-to-end arguments in system design. *ACM Transactions on Computer Systems 2,* 4 (Nov.), 277.

Sharp, C., Schaffert, S., Woo, A., Sastry, N., Karlof, C., Sastry, S., and Culler, D. 2005. Design and implementation of a sensor network system for vehicle tracking and autonomous interception. In *Proceedings of 2nd European Workshop on Wireless Sensor Networks (EWSN)*.

Simon, G., Maróti, M., Lédeczi, A., Balogh, G., Kusy, B., Nádas, A., Pap, G., Sallai, J., and Frampton, K. 2004. Sensor network-based countersniper system. In *Proceedings of 2nd ACM International Conference on Embedded Networked Sensor Systems (SenSys'04)*.

Smyth, A., Pei, J., and Masri, S. 2003. System identification of the vincent thomas bridge based using earthquakes records. *Earthquake Engineering and Structural Dynamics 33*.

Stann, F. and Heidemann, J. 2003. Rmst: Reliable data transport in sensor networks. In *Proceedings of 1st IEEE Workshop on Sensor Network Protocols and Applications (SNPA)*.

Stathopoulos, T., Girod, L., Heidemann, J., and Estrin, D. 2005. Mote herding for tiered wireless sensor networks. Tech. Rep. 58, CENS. Dec. 7.

Stathopoulos, T., Heidemann, J., and Estrin, D. 2003. A remote code update mechanism for wireless sensor networks. Tech. Rep. 30, CENS. Nov. 26.

Szewczyk, R., Mainwaring, A., Polastre, J., Anderson, J., and Culler, D. 2004. An analysis of a large scale habitat monitoring application. In *Proceedings of 2nd ACM International Conference on Embedded Networked Sensor Systems (SenSys'04)*. 214–226.

Tennenhouse, D. L. and Wetherall, D. J. 1996. Towards an active network architecture. *Computer Communication Review 26*, 5–18.

Wan, C. Y., Campbell, A. T., and Krishnamurthy, L. 2002. Psfq: A reliable transport protocol for wireless sensor networks. In *Proceedings of 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*. ACM.

Woo, A. and Culler, D. 2003. Evaluation of efficient link reliability estimators for low-power wireless networks. Tech. Rep. CSD-03-1270, University of California, Berkeley. Apr.

Xu, N., Rangwala, S., Chintalapudi, K., Ganesan, D., Broad, A., Govindan, R., and Estrin, D. 2004. A wireless sensor network for structural monitoring. In *Proceedings of 2nd ACM International Conference on Embedded Networked Sensor Systems (SenSys'04)*.

Yarvis, M., Kushalnagar, N., Singh, H., Rangarajan, A., Liu, Y., and Singh, S. 2005. Exploiting heterogeneity in sensor networks. In *Proceedings of 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'05)*.