The Tenet Architecture for Tiered Sensor Networks

Omprakash Gnawali* Ben Greenstein†‡ Ki-Young Jang* August Joki† Jeongyeup Paek*‡
Marcos Vieira* Deborah Estrin† Ramesh Govindan* Eddie Kohler†
Center for Embedded Networked Sensing
*University of Southern California †University of California, Los Angeles
http://tenet.usc.edu/

Abstract

Most sensor network research and software design has been guided by an architectural principle that permits multi-node data fusion on small-form-factor, resource-poor nodes, or *motes*. We argue 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. Masters provide increased network capacity. Tenet constrains multi-node 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.

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 Engineering**]: Reusable Software
General Terms: Design, Experimentation, Performance
Keywords: Sensor networks, network architecture, 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 multi-node data fusion on resource-constrained sensor nodes as close to the data sources as possible. Like active networks [36], 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. [6, Section 2]

This principle has governed the design of data-centric routing and storage schemes [14, 28], sensor network databases [23], programming environments that promote active sensor networks [19], and a flexible yet narrow-waisted networking stack [26].

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 must implement sophisticated application-specific routing schemes, and algorithms for multi-node fusion, while contending with mote-tier resource constraints.

We examine here a different point in the space of possible architectures, motivated by a property common to many recent sensor network deployments [2, 10, 35]. These deployments have two *tiers*: a lower tier consisting of motes, 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* (Figure 1). 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.

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 multi-node 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 capabili-

‡Contact authors.

This material is based in part upon work supported by the National Science Foundation under Grant Nos. 0121778, 0435497, and 0520235. 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. Marcos Vieira was supported in part by Grant 2229/03–0 from CAPES, Brazil.

SenSys '06, November 1–3, 2006, Boulder, Colorado, USA.

ties 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 paper 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 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 re-task 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 multi-node 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 form:

```
Sample(1000ms, 1, REPEAT, 1, ADC10, TEMP)
   -> ClassifyAmplitude(50, 1, TEMP, ABOVE) -> SendPkt()
```

A *task library* implements a collection of composable tasklets. A reliable multi-tier *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 Tmotes in the mote tier 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 to vibration monitoring 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 multi-node data fusion for efficiency [31]. 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 multi-node fusion to reduce redundant evader reports [31]; our Tenet implementation exhibits higher accuracy and lower overhead that mote-PEG at the cost of marginally higher evader detection latency.

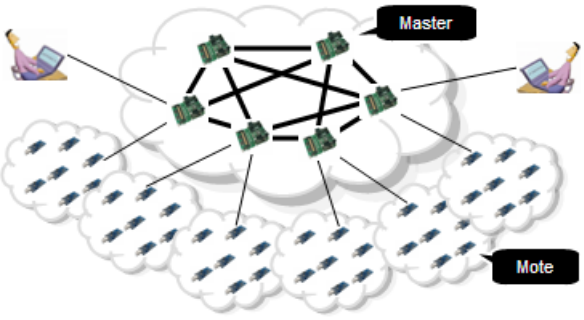

Figure 1—A tiered embedded network. Note that tiering does not imply physical clustering; some motes can communicate with more than one master, possibly over multiple mote hops.

Our evaluation of the individual components of Tenet reveals that Tenet's task library can support high task throughput and 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 in prohibiting in-mote multi-node sensor data fusion. In contrast to a mote-application-specific view of sensor networking, the principle constrains the processing of data received from multiple sensors to be placed at the master tier, regardless of whether that 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 several 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 tradeoff is, of course, a potential loss of efficiency. This loss is limited, however, since Tenet allows motes to process *locally-generated* sensor data. This is crucial—it is infeasible to collect timeseries data from every sensor in a large sensor network—and also covers much of the efficiency gain we have achieved from fusion in prior deployments. This loss can come in three forms. The first is the opportunity cost of in-mote multi-node fusion. We argue that this cost is small, since for most applications that we could think of, there is significant *temporal* redundancy (which we can eliminate using local processing), but little *spatial* redundancy (in almost all deployments, we undersample spatially). 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. 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 4, 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 for the foreseeable future. Even if the technology were to evolve to the point where mote constraints did not matter, we believe our Tenet implementation would still be a viable, flexible, programming system for a distributed system of sensors.

3 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.

3.1 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 that will show when violating the principles is necessary for network performance or functionality.

The Tenet principle prohibits multi-node 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 multi-hop) physical-layer connectivity between them; any master can task any mote as long as there is (possibly multi-hop) physical-layer connectivity between them; and any mote should be 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 this 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 below.

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 that 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.

3.2 Tasks and the Task Library

The key tradeoff faced when developing a language for describing tasks 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, 1, ADC0, LIGHT) -> SendPkt()
```

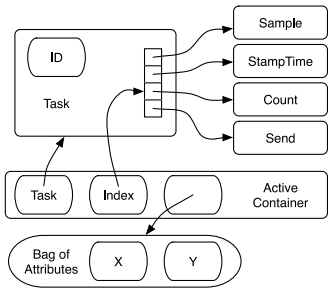

Figure 2—Mote data structures for a typical task

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, and a tasklet may, depending on its parameters, ignore certain inputs, implementing a poor man's conditional.

Tenet's task library was inspired by our own prior work on SNACK [8] and VanGo [9], and somewhat resembles other sensor network tasking architectures [22]. It can also be seen as a particular instantiation of a virtual machine [17]. 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.

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 MeasureHeap tasklet collects statistics on dynamic memory usage in the system, while the Alarm tasklet delays subsequent tasklets on a task chain until a particular time (relative to a global time base) is reached.

The difficulty in constructing such a library is in determining what the right building blocks should be. The GetRoutingInfo tasklet should, of course, record a node's routing state. But what should Sample do? Should it know how to be periodic? Should it know how to sample from more than one ADC channel simultaneously?

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, collect blocks of samples, and sample several channels concurrently, 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.)

The Task Data Structure Figure 2 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, modify, 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 that active container to the first tasklet.

An active container corresponds to 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 array of tasklets. A repeating tasklet clones the active container after each iteration. For example, while executing the task:

```
Wait(1000ms, REPEAT) -> StampTime(TIME, LOCAL)
   -> SendPkt()
```

an active container is cloned and emitted from the Wait 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 $\langle tag, length, value \rangle$. For example, in the above task, the StampTime tasklet adds an attribute with the tag `TIME` to the bag.

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

```
Sample(10000ms, 1, REPEAT, 1, ADC0, LIGHT)
    -> ClassifyAmplitude(39, 1, LIGHT, ABOVE)
    -> StampTime(TIME, LOCAL)
    -> LinkAttributes(LIGHT, TIME, AND) -> SendPkt()
```

will send timestamped light values only when their values are greater than 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.

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 Wait tasklet delays a task for a period of time. Its corresponding wait *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.[1]

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.

[1] 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
ApplyClassification	Reapplies a previous classification	120	0
ClassifyAmplitude	Decides if signal's amplitude is interesting	282	0
CopyAttribute	Copies an attribute	150	0
Count	Increments a counter	136	0
DeleteAttribute	Deletes an attribute	204	0
Disjunct	Helper tasklet for OR-based classification	124	0
GatherStatistics	Calculates statistics on a data set	514	0
Lights	Turns on LEDs based on a given input value	234	0
LinkAttributes	Classify attributes jointly	250	0
MemoryStats	Measures the heap	214	0
NextHop	Generates routing information	94	0
Reboot	Reboots the mote	60	0
Sample	Samples specified ADC channel(s)	4796	113
SampleRssi	Uses radio RSSI as a virtual sensor	208	14
SaveClassification	Saves a previous classification	212	0
SendPkt	Send using packet-based transport	1078	140
SendStr	Send using stream-based transport	1016	142
SetAlarm	Delays a task until a specified time	330	16
StampTime	Reports the current (local or global) time	176	0
Wait	Delays a task (possibly periodically)	950	28

Figure 3—Tasklets in the Tenet task library, with ROM and RAM bytes saved by removing each tasklet from our mote application.

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 that task. Each tasklet encodes its name (from a well-known enum) and its parameters as a tag-length-value attribute.

Motes accept three operations on tasks: installation, modification, 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. A description containing a currently installed task is interpreted as a request to modify that task; when the description is also empty, the mote interprets the message 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.

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 [11], allowing required tasklets to be fetched from a master dynamically.

Examples Figure 3 describes Tenet's current tasklets and their contributions to program size and static memory allocation. 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:

```
Wait(1000ms, REPEAT) -> Count(4, A) -> Lights(A, 0, 0x7)
```

Every second, this task adds 4 to a counter, places the value of this counter into an attribute called *A*, and displays *A* as a pattern of LEDs. (The second and third parameters of Lights determine which byte and bits of *A*, respectively, should be displayed.) *CntToLedsAndRoutedRfm* is a more complicated task to help verify that there is end-to-end connectivity from a master to a mote.

```
Wait(1000ms, REPEAT) -> Count(1, A)
    -> Lights(A, 0, 0x7) -> SendPkt()
```

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:

```
Wait(1000ms, REPEAT) -> NextHop(A) -> SendPkt()
Wait(1000ms, REPEAT) -> MemoryStats(B) -> SendPkt()
```

Ping reports the routing table's next hop information every second. *MeasureHeap* reports a mote's current and peak 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() task 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 properly receiving, installing, and executing the *Reboot* task.

The tasklet Sample serves as the data sources 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 below). For example,

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

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 multi-hop transport.

Sample's parameters make it fairly flexible. The following configuration, for example, samples both channels 0 and 1 and stores five samples of each before sending them:

```
Sample(1000ms, 5, REPEAT, 2, ADC0, A, ADC1, B)
   -> SendPkt()
```

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

```
Sample(1000ms, 10, REPEAT, 1, ADC0, A)
    -> StampTime(B, LOCAL)
    -> ClassifyAmplitude(45, 3, A, ABOVE)
    -> GatherStatistics(A, C, MEAN_DEV, 1)
    -> Lights(A, 1, 0x7) -> SendPkt()
```

Every second, this task takes a sample from the ADC, and passes samples through the chain 10 at a time. On each pass through the chain, the task records the local time, classifies the samples as interesting if at least three of them have an amplitude of at least 45, measures the running mean deviation from the mean, and displays on the LEDs a pattern representative of the values of the samples. Then it transmits the measured statistic and the timestamp. The sample itself is transmitted as well, but only if it happens to be interesting.

3.3 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.

Function	Mechanism
Disseminating tasks from a master to a mote	Tiered reliable flooding of a sequence of packets from any master
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 acknowledgements 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 acknowledgements

Figure 4—Tenet networking mechanism summary

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 5). As a result, we had to build a complete networking 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 multi-hop 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 or the number of masters, whichever is greater. Intuitively, this is the best the routing system can do, short of implementing source routing (a possibility for future work).

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, the data may be irretrievably lost.

The following sections describe the design of Tenet's networking subsystem and how the design achieves these goals; Figure 4 summarizes its novel mechanisms.

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; our experiments use static routing, but Tenet can easily accommodate other wireless routing protocol implementations such as Roofnet [1].

Tenet's addressability also calls for any mote to be able to return a response to the tasking master. Standard tree-routing protocols do not suffice for this 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 border master periodically sends beacons into the mote cloud. When a mote receives a beacon, it relays that 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 requires mote state proportional to the number of active masters, and as long as a mote can hear at least one master, it can send traffic to the master tier. We have modified two standard tree routing protocols (MultiHopLQI and MintRouting; we use the former in our experiments) 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 that to an IP address (by prepending its own 16-bit subnet mask to that address), then determines the next hop towards that IP address using the IP routing table. It then encapsulates the packet in a UDP packet, and sends that 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 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 below) 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 re-task 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 below. Observe that, 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 uses this observation. When a mote gets a task response data packet from a non-parent, it establishes a route entry to the source address (say S) in the packet, with the next hop set to the sender. It also sets a timer on the route entry (in our implementation, 30 seconds); when the timer expires, this entry is removed. If the entry existed previously, the mote resets the associated timer. 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 timer. 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.

Data-driven route establishment maintains one timer and one routing entry per actively communicating mote. More precisely, each mote maintains at most one timer and one routing entry for each active mote in the subtree of its nearest master. There is some scope for optimization; it may suffice to maintain just one timer overall, an approach we have not yet implemented.

Tiered Task Dissemination 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 paper, 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 only those motes whose attributes satisfy the predicate complete execution.

Tenet's reliable task dissemination mechanism is built upon a generic reliable flooding protocol for tiered networks called *TRD*. 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 5). 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 attached motes (in case M happens to be a border master). Motes also cache received packets, and rebroadcast previously unseen packets to their neighbors, and so forth. In our implementation, both master and mote caches are of fixed size (25 entries), and cache entries are replaced using LRU.

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 [18], 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) sequence numbers received from each active master. If the node detects that a neighbor has a higher sequence number than in its own cache, it immediately requests the missing sequence number 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 interject 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 proportional to the number of active masters.

On the master side, TRD is implemented within a separate daemon that also implements the transport protocols described below. 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, TRD assigns it a unique task ID and maintains a binding between a task ID and the application. Applications can use this task ID to modify or delete 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. TRD keeps copies of task packets received in memory. An interesting side effect of TRD's robustness is that a master which recovers after a crash may receive a copy of a task it had previously sent out. TRD can (our current implementation does not) check the received copy against all the tasks listed in its cache, and delete that task if a copy is not found.

Our mote implementation of TRD is engineered to fit within mote resource constraints and 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. Eventually, TRD will need to implement a consistent aging strategy by which master entries in the summary are individually timed out to accommodate new masters. We plan to explore this in future work.

Finally, Tenet applications might also wish to selectively re-task 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 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.

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 three types of delivery mechanisms: a best effort transport useful for loss-tolerant periodic low rate applications, a transactional reliable transport for events, and a stream transport for high-data rate applications (imaging, acoustics, vibration data). All three 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, SendPkt(), SendPkt(E2E_ACK), or SendStr()). The implementation of best-effort transport is conventional; 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 that of TCP. However, because stream delivery is fundamentally simplex, the connection establishment state machine is not as complex 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 4).

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.[2] Furthermore, our implementation has a configured limit (currently 4) on the number of open connections a mote may have. Both stream and transactional transport work transparently across the two tiers in Tenet. Both 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. Our current implementation does not incorporate any congestion control mechanisms, which we have left to future work.

[2]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.

3.4 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 some functionality required to support many sensor network applications.

Infrastructural components can often 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 that into Tenet. Information from infrastructural components (such as location and time) can be exported to applications through tasklets; indeed, we have developed preliminary versions of tasklets that export routing state (next hop), and that allow applications to read global time and synchronize task execution.

In the near-term, we will explore several extensions to the Tenet system. The first is support for actuation, which is 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, support for low-latency communication or bounded-latency messages is important for time-critical sensing applications and will need careful transmission scheduling mechanisms in the networking subsystem.

Longer-term, we will explore the impact of disconnections caused by mobility, mechanisms for ensuring the authenticity and integrity of data, and methods for multi-user access control and resource management.

4 Tenet Evaluation

This section evaluates Tenet through microbenchmarks of its tasking and networking subsystems and reliable stream transport, and through two 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; and that even challenging applications may be implemented as Tenet tasks with little efficiency loss.

4.1 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 26 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(20000ms, 1, REPEAT, 1, ADC0, A) -> SendPkt()
```

Task	Max Concurrency	Memory Usage (bytes) Application	Overhead	Data Transmitted (bytes/packet) Dissemination	Output	% Overhead
Diagnostics	—	122	32	38	26	46%
Sample[1] → SendPkt	38	122	18	24	6	67%
+ StampTime	36	134	20	32	14	57%
+ ClassifyAmplitude	31	150	22	44	14	57%
+ GatherStatistics + DeleteAttribute	26	192	26	62	18	44%
Sample[40] → SendPkt	—	200	18	24	84	5%

Figure 5—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.

We increase the sampling period to 20 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 is a union of *Ping* and *MeasureHeap* plus a timestamp:

```
Wait(1000ms, REPEAT)
   -> StampTime(A, LOCAL) -> MemoryStats(B)
   -> NextHop(C) -> SendPkt()
```

A single Tmote mote can concurrently execute the diagnostics task plus 38 sample-and-send tasks, but the mote has insufficient resources to support a 39th. 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 most complicated task is as follows:

```
Sample(20000ms, 1, REPEAT, 1, ADC0, A)
  -> StampTime(B, LOCAL)
  -> ClassifyAmplitude(0, 1, A, ABOVE)
  -> GatherStatistics(A, C, MEAN_DEV, 1)
  -> DeleteAttribute(A) -> SendPkt()
```

This task is similar to the data acquisition and processing example above, except that we additionally delete the sample for good measure. Figure 5 shows the results: a Tmote can execute concurrently 26 of this most 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 5 shows the total bytes allocated from the heap per task in steady state, and the total number of bytes allocated to manage those heap blocks (two bytes per block). Our Tenet Tmotes have around 5813 bytes available in RAM for the heap and call stack. In the steady state, a sample-and-send task uses 140 $(122+18)$ bytes allocated from the heap and the diagnostics task uses 154 $(122+32)$ bytes. Thus, even ignoring the call stack, a Tmote wouldn't be able to support more than $\lfloor(5813-154)/140\rfloor = 40$ concurrently executing sample-and-send tasks while a diagnostics task is running. In actuality, a Tmote supports 38. On more RAM-constrained motes, this number can be much lower; a Tenet micaZ, for example, supports only 9 sample-and-send tasks.

Input Samples	Execution Time (ms)
1	2.2
40	2.6
400	6.2
800	11.2
1200	14.8

Figure 6—Execution time of measuring the mean deviation from the mean as a function of the number of input samples. Averaged over five runs.

Even tasks that use our most CPU-intensive tasklet, GatherStatistics, experience memory constraints before processor constraints. The following task demonstrates this:

```
Sample(1000ms, 1, REPEAT, 1, ADC0, A)
   -> StampTime(B, LOCAL)
   -> GatherStatistics(A, C, MEAN_DEV, 1)
   -> StampTime(D, LOCAL) -> DeleteAttribute(A)
   -> SendPkt()
```

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 its execution speed. Our hypothesis is that for the tasklets we have written, none consume enough MCU cycles such that concurrency will be bounded by the microcontroller. Figure 6 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 14.8 milliseconds. Thus, a mote running several tasks that gather statistics will only be CPU-bound when the aggregate sampling rate is approximately 81,000 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 5 shows the packet overhead associated with adding type and length fields to each of our data attributes. It requires 24 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 put more than one sample in an attribute. When the master specifies that 40 samples be sent in each packet (the Sample[40] → SendPkt task), this overhead drops to 5%.

4.2 Application Case Study: Pursuit-Evasion

How much, if at all, does Tenet degrade application performance relative to a mote-native implementation that per-

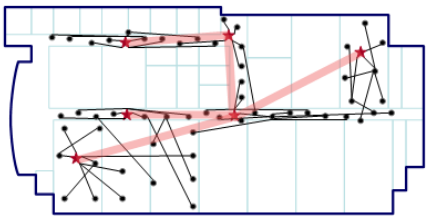

Figure 7—Testbed topology as gathered by NextHop() → SendPkt(). Masters are stars, motes are dots. PEG experiments use only the central master.

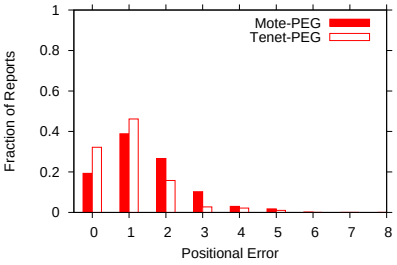

Figure 8—PEG application error

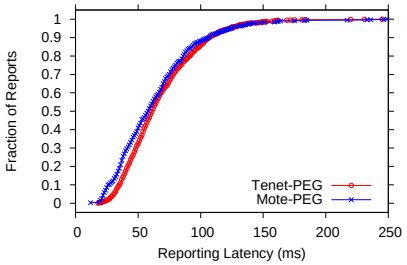

Figure 9—PEG detection latency

forms in-mote multi-node 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 multi-robot coordination. In this paper, 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. [31] 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 re-implemented 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 multi-node data fusion to determine the centroid of all sensors that detect an evader; the other mechanisms route leader reports to pursuers. Our re-implementation, 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 multi-node fusion, it cannot achieve similar efficiencies.

We have implemented a single pursuer, single evader PEG application for Tenet. In Tenet-PEG, the pursuer 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 multi-node 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 re-tasks 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 7. 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 non-interfering channels. 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 [4], and since we only require coarse-grained localization (see below), it is a reasonable choice for our experiments as well. Second, to create a realistic multi-hop topology, we limit the transmit power of each mote. This

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

Figure 10—PEG application overhead

results in a topology with a 9-hop diameter, and is comparable to the diameter of the network used in [31]. 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 [16]. 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 [3]. 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 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 that detection reaches the pursuer. We measure this using FTSP 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-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. Our preliminary analysis indicates that the 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 10 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 multi-node data fusion for performance, Tenet implementations can perform comparably with mote-native implementations. Although some applications may require in-mote multi-node data fusion, our results argue that *starting* with in-mote multi-node data fusion is essentially premature optimization: easier-to-manage Tenet-style versions may perform just as well.

4.3 Application Case Study: Vibration Monitoring

In Section 4.1, 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, but we can give concrete examples. For instance, we have built a Tenet implementation of a mature structural monitoring system, Wisden [25, 39], which reliably transmits vibration samples from a network of motes to a base station. Each mote transmits only interesting events using a simple *onset detector* [25], greatly reducing communication requirements. Wisden implements specialized mechanisms for reliability and time synchronization. An onset is defined to be a part of a signal waveform that exceeds the mean amplitude by more than a few standard deviations.

To port Wisden to Tenet, we implemented a DetectOnset tasklet and used it to build a task functionally equivalent to Wisden's mote code. Specifically:

```
SampleMDA400(20ms, 40, A, Z_AXIS)
   -> DetectOnset(A, B) -> SendStr()
```

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

Figure 11 shows the vibration time-series on two MicaZ motes using a single master. One mote was programmed with the above task, but *without* DetectOnset (the lower timeseries). The other was programmed with the above task (the upper timeseries). We then put the two motes on a table and hit it. The task without DetectOnset 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. 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 judgement is that, even for these reasons alone, Tenet significantly simplifies application development.

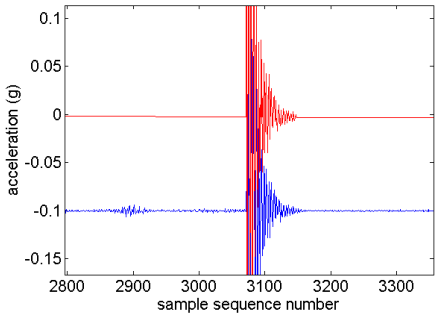

Figure 11—Vibration sensing with onset detector

Masters	1	2	3	4	5	6
Average (ms)	543.60	317.52	225.84	177.88	126.29	116.05
Min (ms)	35.64	37.26	33.42	29.96	28.45	31.08
Max (ms)	871.58	730.53	437.06	364.56	272.62	270.09
Std. deviation	224.05	197.42	128.83	109.18	94.14	77.06

Figure 12—Task dissemination latency using StampTime() → SendPkt()

4.4 Manageability

Network monitoring is an important part of networked system manageability. Although we have not extensively evaluated manageability as such, it 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 any instant:

```
Wait(1000ms, ONCE) -> NextHop(A) -> SendPkt(E2E_ACK)
```

This obtains each mote's routing parent using our reliable packet delivery protocol. Figure 7 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:

```
StampTime(A, GLOBAL)
   -> Wait(1000ms, ONCE) -> SendPkt(E2E_ACK)
```

timestamps a task using the time computed by the FTSP 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 7 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. 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).

4.5 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:

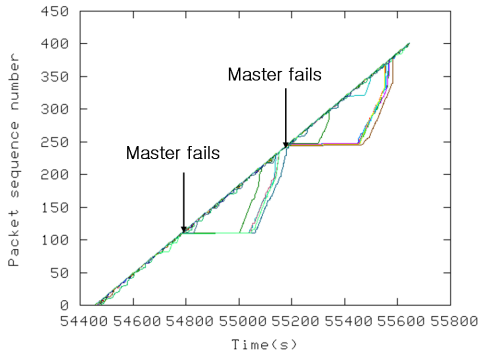

Figure 13—Sequence number evolution of stream transport connections

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

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

Figure 13 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 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 negative acknowledgement rates 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).

5 Related Work

In the earliest work on sensor network architecture, Estrin et al. [6] motivate the need for application-specific multinode aggregation within the network. More recently, Culler et al. [5] 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 [26]. Tenet is complementary to SNA, since Tenet constrains the placement of application functionality in a tiered system and does not address the modularization of software. The mote tier in Tenet can be implemented on an SP-based system.

The Tenet principle shares some similarities with the Internet's end-to-end principle [30]. 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).

Many sensor network deployments in the recent past have been tiered. Examples include the Great Duck Island deployment [35], the James Reserve Extensible Sensing System [10], and the Extreme Scaling Network [2]. In these deployments, tiering provides greater spatial scale and increased network capacity. Other systems have been built upon tiered networks. SONGS [21] focuses on a set of high-level services designed to extract semantic information from a tiered network. Lynch et al. [15] discuss tiered networks for structural monitoring since upper-tier nodes can perform the sophisticated signal processing functions required for the application. Rhee et al. [29] 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 [20] 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. [24] describe a similar result for the lifetime of a tiered network. Finally, Yarvis *et al.* [40] 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 five years. We now discuss these.

Mate [17, 19] 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, could be represented as a single bytecode instruction provided by an application-specific instruction set. This would greatly reduce the overhead in disseminating new applications and would simplify application construction. The tradeoff—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 that program can contain multiple threads. Tenet motes explicitly support concurrent and independent application-level tasks. In that respect, Tenet also differs from tasks in SHARE [22], 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 [12] operating system and is written in nesC [7]. The design choices of TinyOS—pushing as many decisions as possible to compile-time—has 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 [8].

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 employs similar techniques (exponential timers and suppression) as prior code dissemination protocols [13, 18, 33]. 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 meta-data summaries used for loss recovery are specific to code pages.

Reliable data transport has received some attention in the literature. RMST [32] (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 [37] (Pump Slowly, Fetch Quickly) is a hop-by-hop reliable transport protocol designed for sensor network reprogramming, Wisden [39] is a system for reliably transporting structural vibration data from a collection of sensors to a base station, and DataRel [34] provides a TCP-like abstraction for transporting data from a mote to a border master in a tiered network. By contrast to these systems, Tenet's reliable transport ensures point-to-point delivery of processed sensor data to any master in a tiered network.

Most prior routing protocols either support data delivery to a base station [27, 38] or to multiple sinks [10]. Some, such as Centroute [34], 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.

6 Conclusions

In this paper, 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.

Acknowledgements

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.

References

[1] The MIT Roofnet Project. `http://pdos.csail.mit.edu/roofnet/`.

[2] A. Arora et al. *ExScal*: Elements of an extreme scale wireless sensor network. In *Proc. 11th IEEE International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA '05)*, August 2005.

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

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

[5] D. Culler et al. Towards a sensor network architecture: Lowering the waistline. In *Proc. 10th Hot Topics in Operating Systems Symposium (HotOS-X)*, pages 139–144, June 2005.

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

[7] D. Gay et al. The nesC language: A holistic approach to networked embedded systems. In *Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–11, June 2003.

[8] B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (SNACK). In *Proc. 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, pages 69–80, Nov. 2004.

[9] B. Greenstein et al. Capturing high-frequency phenomena using a bandwidth-limited sensor network. In *Proc. 4th ACM Conference on Embedded Networked Sensor Systems (SenSys '06)*, Nov. 2006. To appear.

[10] R. Guy et al. Experiences with the Extensible Sensing System ESS. Technical Report 61, CENS, Mar. 29 2006.

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

[12] J. Hill et al. System architecture directions for network sensors. In *Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104, Nov. 2000.

[13] J. Hui and D. Culler. The dynamic behavior of a data dissemination algorithm at scale. In *Proc. 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, Nov. 2004.

[14] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. 6th Annual International Conference on Mobile Computing and Networking (MobiCom 2000)*, pages 56–67, Aug. 2000.

[15] V. A. Kottapalli et al. Two-tiered wireless sensor network architecture for structural health monitoring. In S.-C. Liu, editor, *Proc. SPIE 5057—Smart Structures and Materials 2003: Smart Systems and Nondestructive Evaluation for Civil Infrastructures*, Aug. 2003.

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

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

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

[19] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Proc. 2nd Symposium on Networked Systems Design & Implementation (NSDI '05)*, May 2005.

[20] B. Liu, Z. Liu, and D. Towsley. On the capacity of hybrid wireless networks. In *Proc. 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Infocom 2003)*, Mar. 2003.

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

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

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

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

[25] J. Paek, K. Chintalapudi, J. Cafferey, R. Govindan, and S. Masri. A wireless sensor network for structural health monitoring: Performance and experience. In *Proc. 2nd IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, May 2005.

[26] J. Polastre et al. A unifying link abstraction for wireless sensor networks. In *Proc. 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys '05)*, pages 76–89, Nov. 2005.

[27] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proc. 4th International Symposium on Information Processing in Sensor Networks (IPSN '05), SPOTS track*, Apr. 2005.

[28] S. Ratnasamy et al. GHT: A geographic hash table for data-centric storage. In *Proc. 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, pages 78–87, Sept. 2002.

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

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

[31] C. Sharp et al. Design and implementation of a sensor network system for vehicle tracking and autonomous interception. In *Proc. 2nd European Workshop on Wireless Sensor Networks (EWSN)*, Jan. 2005.

[32] F. Stann and J. Heidemann. RMST: Reliable data transport in sensor networks. In *Proc. 1st IEEE Workshop on Sensor Network Protocols and Applications (SNPA)*, May 2003.

[33] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report 30, CENS, Nov. 26 2003.

[34] T. Stathopoulos, L. Girod, J. Heidemann, and D. Estrin. Mote herding for tiered wireless sensor networks. Technical Report 58, CENS, Dec. 7 2005.

[35] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler. An analysis of a large scale habitat monitoring application. In *Proc. 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, pages 214–226, Nov. 2004.

[36] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2):5–17, Apr. 1996.

[37] C. Y. Wan, A. T. Campbell, and L. Krishnamurthy. PSFQ: A reliable transport protocol for wireless sensor networks. In *Proc. 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, Sept. 2002.

[38] A. Woo and D. Culler. Evaluation of efficient link reliability estimators for low-power wireless networks. Technical Report CSD-03-1270, University of California, Berkeley, Apr. 2003.

[39] N. Xu et al. A wireless sensor network for structural monitoring. In *Proc. 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, Nov. 2004.

[40] M. Yarvis et al. Exploiting heterogeneity in sensor networks. In *Proc. 24th Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Infocom 2005)*, Mar. 2005.