The Tenet Architecture for Tiered Sensor Networks

Tenet Architecture Overview
- Two-tiered wireless sensor network architecture consisting of masters & motes
- Based on the Tenet Principle
  - Motes process locally-generated sensor data
  - Masters perform multi-node data fusion
  - Cost and complexity of an application specific system outweigh its benefits

Tenet Architecture
- Masters are relatively unconstrained 32-bit nodes
- Motes are resource/processing constrained nodes with a specific capability library
- Masters “task” motes to perform actions from their capability library
- Motes execute tasks and send “task responses” back to masters
- Essentially implements a virtual machine

Tenet Principle
- Prohibits in-mote multi-node sensor data fusion
- All multi-node data fusion performed on masters
  - More processing power and storage allow for more sophisticated fusion
  - Gather data from larger spatial area resulting in more data, more accurate decisions
- Potential for efficiency loss
  - Opportunity cost of in-mote data fusion
  - Distance data has to travel
  - Increased network congestion

Tenet Tasks
- Asymmetric task communication – only masters can task motes
  - Tasks are composed of a sequential series of functions called “tasklets”
  - Motes support a limited number of tasklets in their “task library”
  - Motes can execute many tasks concurrently
  - Upon a task’s completion motes return data/results through a task response
Task Example

- Application wants to be notified when the temperature exceeds 50 degrees
  
  `Sample(1000ms, 1, REPEAT, 1, ADC10, TEMP) -> ClassifyAmplitude(50, 1, TEMP, ABOVE) -> SendPkt()`
  
- Sample and ClassifyAmplitude are tasklets
- ADC10 is the A/D converter that the temperature sensor is connected to
- Every 1000 ms the A/D converter will be read and the value will be stored in TEMP
- If TEMP is 'ABOVE' 50 then a task response is sent back to the application

Task Structure

- Task object holds task id and tasklet chain
- An active container holds state information and associated data for a task
- Active container moves its way down the tasklet chain as each one executes
- Once all tasklets are complete the active container is deleted

More Task Information

- Tasklets commanded to repeat do so by creating a new active container every time they finish executing
- Tasks can have more than one active container at a time
- After an active container is created on a mote it is placed in a queue
- The Task Scheduler cycles through this queue executing the current tasklet in each active container
- Allows for multiple tasks to be executing concurrently on a mote

Task Operations

- Tasks are sent to masters in the form of a Task ID and a list of tasklets
- Tasklet parameters are passed as tag-length-value attributes
- Three possible task operations:
  - Installation – Mote receives a Task ID with a tasklet list that is not currently running on its system
  - Modification – Mote receives a Task ID with a tasklet list that is already running on its system
  - Deletion – Mote receives a Task ID without a tasklet list that is already running on its system

Network Subsystem

- Two main responsibilities
  - Sending Tasks to motes
  - Sending Task Responses to masters
- Tasks are broadcast to all motes
- Individual motes send task responses to a specific master
- Masters send ACKs back to a specific mote after receiving a task response

Routing

- All nodes have globally unique 16-bit identifiers
  - On motes ID is the 16-bit TinyOS node identifier
  - On masters ID is the lower 16 bits of their IP address
- Four routing cases:
  - Master to master (IP routing)
  - Master to mote (Data-driven routing)
  - Mote to mote
  - Mote to master (Tiered routing)
Task Dissemination

- Master floods task descriptions to all motes
  - Tasking motes is an infrequent event
  - Applications usually task most, if not all, motes
  - Subsets of motes can be tasked by adding a predicate tasklet to the task description
- Use generic reliable flooding protocol for tiered networks called TRD
  - Provides the abstraction of reliably flooding a sequence of packets from any master to all motes

TRD Example

- Master M wants to transmit a task packet
  - TRD locally caches a copy of the packet on M and assigns it a sequence number
  - Packet is then broadcast to any attached neighbor masters and motes
  - Motes cache received packets and rebroadcast previously unseen packets to their neighbors
    - Packets are cached in flash
    - Master and mote caches are of fixed size
    - Cache entries are replaced using LRU

TRD Continued

- Occasionally a node may not receive a broadcasted packet due to wireless transmission errors
  - Each node periodically transmits a summary of the packets it has in its cache
  - Summary contains the last $K$ sequence numbers received from each active master
  - If a node detects that another node has a higher sequence number it requests the missing packet(s) from that node
- When a node boots up it broadcasts its empty packet summary

Task Responses

- Motes transmit task responses to the nearest master
  - Path to the nearest master is always through a mote’s parent node
  - Master’s periodically send beacons into the mote cloud
  - Motes retransmit received beacons to their neighbors along with an associated path metric
  - Motes select their parent based on who provides the best path to a master
  - Packets received by motes not from their parent are always then forwarded to their parent
  - Master then forwards the response to the appropriate master

Task Response ACKs

- Masters have to be able to send ACKs & NACKs to a specific mote who sent a task response
  - Done using data-driven route establishment mechanism
    - If a mote receives a task response from a non-parent it creates a route entry to the source address
    - A timer is set on the route entry establishing the entry’s time-to-live
    - Every time the entry is used its timer is reset
  - Route from a master to a specific mote exists as long as the mote has recently communicated with the master

Parent Node Determination
Task Dissemination

- Wants to send Task A
- Wireless transmission error
- Forwarded throughout the mote cloud
- Forwarded into the mote cloud

Missed Task Transmission

- Node X wants to send Task A
- First sent in the Master Tier
- Forwarded into the mote cloud
- Wireless transmission error
- Node Y periodically transmits cached packet summary
- Node Z realizes it missed Task A
- Requests Task A packet from Y
- Task A packet is forwarded to node Z

Task Response Transmission

- Node Z completes Task A and needs to send a response
- Node Z sends task response to its parent node
- Node Y sends the packet to the master tier

Task Response ACK

- Master sends task response ACK
- Response is forwarded to appropriate master in the master tier
- Node Y forwards the packet and resets the timer in the routing table
- Node X forwards the packet and resets the timer in the routing table

Reliable Transport

- Three types of delivery mechanisms
  - Best effort transport
    - Conventional implementation
    - Useful for loss-tolerant periodic low rate applications
  - Stream transport
    - Establishes end-to-end connection between a master and a mote
      - Connection establishment approach similar to TCP
      - Requires less handshaking for connection setup/teardown
      - Hop-by-hop retransmission of lost packets
      - Useful for high data rate applications
  - Transactional reliable transport
    - Uses stream transport connection establishment
    - Useful for sending events

Evaluation

- Task Concurrency
  - Tmotes can support between 26 and 38 concurrent tasks, depending on the task’s complexity
  - Resource bottleneck: memory
- Data transmitted and packet overhead
  - Data attributes tag and length fields add significant packet overhead

<table>
<thead>
<tr>
<th>Task</th>
<th>Max Concurrence</th>
<th>Max's Task (bytes)</th>
<th>No. Transmitted</th>
<th>No. Received</th>
<th>Overall % Received</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transactional</td>
<td>3</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>100%</td>
</tr>
<tr>
<td>Stream</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>100%</td>
</tr>
<tr>
<td>Best Effort</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>100%</td>
</tr>
</tbody>
</table>
Tenet Architecture vs. App Specific

- Implemented a pursuit evasion game (PEG) on two platforms
  - Tenet-PEG: uses Tenet Architecture
  - Mote-PEG: uses conventional architecture

Tenet vs. SP

- Both the Tenet Architecture and the Sensornet Protocol (SP) attempt to simplify wireless sensor network development by addressing the lack of an overall sensor network architecture
- The Tenet Architecture is an entire sensor network architecture while SP is just a building block to be used in a sensor network architecture
- SP provides greater growth potential in that it independent of the specific link protocol being used

Critique

- Better conditional statement support in task library
- Paper states that TRD can only support 5 masters but the PEG implementation used 6
- Data driven route establishment currently maintains one timer for each active entry. Could be optimized to only use one overall
- What happens when a master attempts to task a mote and the mote does not accept the new task?
- No mention of power consumption characteristics / expected lifetime of the motes

Questions