Implementation of Trajectory Based Forwarding on TinyOS

Ashwin Kashyap (ashwink@paul.rutgers.edu) Andrew Tjang (atjang@eden.rutgers.edu) Michael Pagliorola (mtp42@remus.rutgers.edu) Dragos Niculescu (dnicules@cs.rutgers.edu) Badri Nath (badri@cs.rutgers.edu)

Introduction

This document describes the design and implementation of TBF on the Mica Motes. Motes are tiny embedded devices with software radio and a slew of other sensors. They are highly resource constrained, having just 128K of programmable memory and 4K of RAM. TinyOS is an operating system designed for these embedded devices. It is a very simple OS, consisting of a non-preemptive round robin scheduler and the accompanying drivers needed to use the radio and the sensor devices.

TBF is a protocol, in which nodes in a sensor network (we assume that the nodes know their position by some means) can send packets along arbitrary trajectories, represented by mathematical expressions. TBF can be used in implementing important networking functions such as flooding, discovery, and network management. Trajectory routing is very effective in implementing many networking functions in a quick and approximate way, as it needs very few support services. Once TBF is deployed on to any network, packets can be routed along any trajectory specified. The applications of such a versatile forwarding algorithm are limitless, especially in disposable networks where nodes are thrown or dropped to form a one-time use network.

Overview of TBF

There are many instances when standard bootstrapping or configuration services are not available. In such cases, TBF will serve as a competent substitute. Forwarding based on trajectories decouples the path name from the path itself. Since a trajectory can be specified independently of the nodes that makeup the trajectory, routing can be very effective even when the intermediate nodes that makeup the path fail, or are otherwise unavailable. The specification of the trajectory can be independent of the ultimate destinations of the packet, and thus provides efficient methods for implementing network functions such as flooding, discovery and multicast. In the end, we want to be able to specify any routing trajectory and have packets forwarded to all nodes along that trajectory.

TinyOS networking stack

At the 10,000-mile overview, the TinyOS "messaging" system is based on the Active Message framework. This fits rather nicely with an event-driven operating system. Anytime a message is received, an event is generated, and an appropriate handling method is called. Message packets are encapsulated with the TOS_Msg structure (with

a default size of 36 bytes). TOS_Msg buffers are stored in the TOS frame, and thus are globally accessible, which optimizes memory usage.

Sending

The process is totally asynchronous and the control is immediately returned to the caller, another event called the send_done event will be triggered on completion. When the message layer accepts a send command, it owns the message buffer, and the requesting component should not access the buffer until the send_done event is called.

Receiving

The component can register a handling function for every type of received message, this is typically done during compile time by "wiring" the appropriate even handler to the appropriate component method. When a message arrives, it fills the buffer structure, and the active message layer decodes the handler type and dispatches. The layer gives the processing component the pointer to the buffer, and expects a pointer to a blank buffer upon return.

The GENERIC_COMM component provides a more in depth view of the network stack. GENERIC_COMM contains six additional components, from the high-level messaging component all the way down to modulation of individual bits on the radio link. More information about the network stack and GENERIC_COMM can be found in Lesson 4 of the TinyOS tutorial.

Representing and encoding trajectories

We need to represent the trajectory in a compact form, yet it should be expressive so that we are able to represent equations of arbitrary complexity (provided the packet is large enough). It should also be easy to encode and decode, and the process must be fast. Ideally the encoding and decoding must be orders of magnitude faster than the time it takes to send out a packet. The solution we propose is to represent the trajectory as a parametric equation, that is, we have an equation for x(t) and y(t). The trajectory is encoded in Reverse Polish Notation (RPN) also known as the postfix notation. We chose RPN after evaluating several other choices like active message packets and infix notation. The encoding is binary in order to save on packet size. Using schemes like XML might be very convenient and portable, however they are not compact enough to be sent over TOS messages, which have just 30 bytes as payload. The size of the trajectory encoding depends on the complexity of the equation to be represented. Many common equations like sine waves, straight lines and circles are easily encoded and are compact enough to be transmitted over TOS messages.

A note about RPN

This is a formal logic system that enables us to represent arbitrary mathematical expressions without using parenthesis. For example:

[infix notation] (4 + 5) * 6 could be expressed as:

[prefix notation] * + 4 5 6

[postfix notation] 6 4 5 + *

The advantage of using RPN over prefix notation is that we can evaluate arbitrarily complex equations just with the help of a stack. In order to evaluate RPN expressions, we simply read from left to right. If the character under consideration is an immediate (a number) then we push it onto the stack. If we read a function symbol (like *, + or sin) we pop the required number of operands from the stack and call the function on these operands. This is extremely simple, evaluating prefix or infix expressions is significantly more complex than this.

Lookup codes

We need to send the resulting equations over the packet; one of the choices is to send the equation in human readable text format. There are certain problems with this:

- Size of the resulting encoding is large
- We need to parse integers and floating point numbers which can be slow

Instead, we have lookup codes that are just 1 byte long and represent the functions in the expression. Currently, we can have three different data types for immediates:

- Character (1 byte)
- Integer (2 bytes)
- Floating point (4 bytes)

Each of these are encoded in the little endian format, we further prefix a code that indicates what type of data the following immediate is of. Our representation is compact and we do not need to parse any data bytes in order to decode immediates.

This is the current list of functions that we support:

Code	Symbol name	Pointer to the function
Immediates		
1	FLOAT	NULL
2	INT	NULL
3	CHAR	NULL
Functions with one argument		
4	SIN	sin
5	COS	cos
6	TAN	tan
7	ASIN	asin
8	ACOS	acos

9	ATAN	atan
16	LOG10	log10
17	LOG	log
18	EXP	exp
19	SQRT	sqrt
20	CEIL	ceil
21	FLOOR	floor
22	MY_PI	NULL
Functions with 2 two arguments (code >128)		
129	ADD	my_add
130	SUB	my_sub
131	MUL	my_mul
132	DIV	my_div
133	POW	pow
Code to represent the parameter		
255	T	NULL

»

The above table is stored in every Mote that runs TBF. During encoding, we replace all instances of the function string with code (the left most column), immediates parsed into little endian format and prefixed with the appropriate code, the program decides what data type to use in order to encode the trajectory in a compact form. On reception of a packet, the code present in the packet is looked up in the table and the appropriate function is called (the right most column contains pointers to the function). In case the code represents an immediate, the appropriate number of bytes are read from the packet and converted to the endianess of the current architecture.

Examples

Consider 4*sin(t)

The binary postfix encoding will be:

0x03	0x04	0xff	0x04	0x83
char encoding	4	t	sin	*
Total Size: 5 bytes				

Consider 20000*sin(t)

The postfix encoding will be:

0x02	0x204e	0xff	0x04	83
int encoding	20000	t	sin	*
Total Size: 6 bytes				

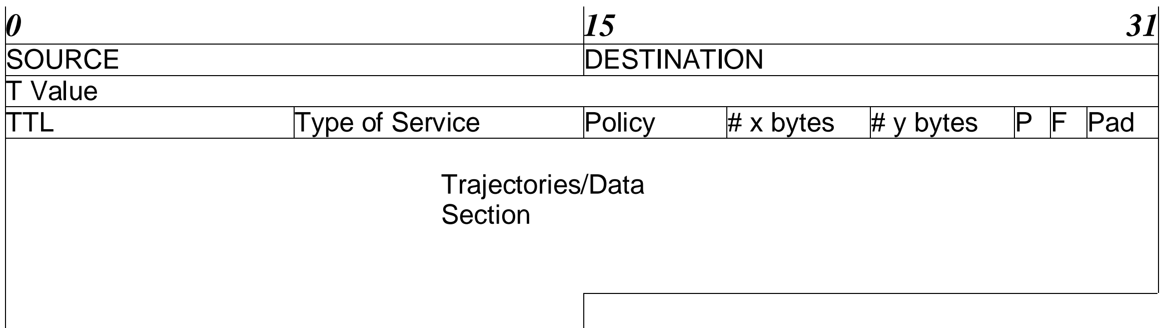
Consider 3.323*t + 90

The postfix encoding will be:

0x01	0x08ac5440	0xff	0x83	0x03	0x5a	0x81
float enc.	3.323	t	*	char enc.	90	+
Total Size: 10 bytes						

Packet format and processing

The TBF packet structure is nested into the data section of a TinyOS packet. Since the default TinyOS packet size is 36 bytes (with 4 bytes for the TOSMsg header and 2 bytes for the CRC), the TBF packet is limited to 30 bytes. This seems to be sufficient for specifying many simple trajectories with space left for data.



TBF header structure

Source - The originating node of the TBF message (2 bytes).

Destination - The final destination (2bytes, not to be confused with the TOS destination, which is the next hop destination)

T-value - Best described by the t value in a parametric function. This allows nodes to know where they are on a particular curve, and in which direction the message is traveling (4 bytes).

TTL - Time To Live field (1 byte).

TOS - Type Of Message, used to distinguish among different types of TBF packets, requesting different services (1 byte).

Policy - The forwarding policy to use (4 bits)

#x/#y bytes - The number of bytes used to encode the x/y parametric equations, respectively (4 bits each)

P - bit specifying if the packet should be processed (1 bit).

F - bit specifying if the packet should be forwarded (1 bit).

The remaining 18 bytes are used to encode the parametric trajectories and any other data.

Neighbor discovery and maintenance

Each node in the network maintains a list of it s immediate neighbors and all forwarding decisions are made by considering the positions of all the neighbors relative to the trajectory. Every node in the network broadcasts a special beacon packet periodically. Any node that receives the beacon will update its neighbor table.

Beacons

Beacons have the following packet structure:

2 byte Moteid	4 byte X coordinate	4 byte Y coordinate	1 byte total_neigh	List of neighbors (up to total_neigh)

Moteid specifies the ID of the mote transmitting the beacon

X/Y-coordinate specifies the location of the current

total_neigh specifies the total number of neighbors this node has

Rest of the packet will have the Ids of all the neighbors of this node

When a node receives the beacon packet, it adds the Moteid, and the locations into the neighbor table. The rest of the data is currently ignored, it is however used by the GUI to display the connectivity graph of the network.

The structure of the neighbor table is as follows:

Mote id	X coord	Y coord	T value	Distance	Dirty	Clock

It should be noted that this structure is never sent out in the packet, each node maintains this table and it is updated whenever a beacon is received. The T-value and Distance fields are local information; it only makes sense with respect to some given trajectory; only the policy algorithms use these to find the relative position of the node with respect to a trajectory. The Dirty field is not currently used and we plan to have a trajectory-caching algorithm that will greatly reduce the calculation overhead for consecutive packets with the same trajectory. The Clock field is used to remove dead nodes, and maintains soft state of the neighbors.

Removing dead nodes

We envision a network with mobile nodes, where intermediaries can move around in the network at will. Whenever a node moves out of a cell, it should be removed from the neighbor list of all other nodes within its radio range (at some time in the future). Also, if the node dies for some reason, we need to collect all allocated resources in the neighbor table. In order to do this, we use the Clock field to keep track of how often the node has beaconed; a high clock value indicates that the node has not beaconed recently. Whenever the clock value for a particular node exceeds a tunable threshold

value, we delete that node from the table. When a node beacons, we set its clock value to zero and increment the clock value of all other nodes by one.

Software range and negative beacons

While the soft state nature of beaconing is sufficient for most cases, it is not particularly well suited for cases with high mobility. It takes a long time to detect if a node is out of communication range. In order to address this issue, we have the concept of software range. Each node maintains an approximate value for its radio communication range. Whenever a node receives a beacon from another node outside its software communication range, the entry for the node is removed from the neighbor table. Let us consider the case when node X is about to move to a new location (x2, y2) from the current position (x1, y1). Before moving out (while still at (x1, y1)) it sends out a beacon, claiming its position to be at (x2, y2). Now, many nodes around the point (x1, y1) will be able to hear the beacon, they calculate the distance to (x2, y2) and if the distance is greater than the software radio range, they immediately remove the node X from the table. This way, we can forcefully update the table without waiting for a long time.

Forwarding policy

Currently, we have implemented two forwarding policies, the maximum progress policy and minimum distance policy. The maximum progress policy will choose the next neighbor such that it makes maximum progress along the trajectory, it does not consider the distance of the node from the trajectory. This can have some unintended side effects in large networks, due to the greedy nature of the algorithm. The minimum distance policy always chooses the node that is closest to the trajectory, without considering how much progress it would make along the curve. The advantage of this algorithm is that in large networks, packets will not stray away from the trajectory. On the flip side, this is very inefficient in terms of the number of hops that need to be made to reach the final destination.

The core algorithm is implemented in the function build_assoc(). For every neighbor, we calculate the smallest distance to the trajectory and the progress the node would make along the trajectory. This is done by first diving the trajectory (in the vicinity if the current node) into discrete parts. The distance from every neighbor to each of these discrete points is calculated and the shortest one is stored, along with the point. We use the neigh_table data structure described previously to do this (the t and the dist fields respectively in the structure). The policy functions sift through this to pick the node best suited to deliver the packet.

Querying protocol

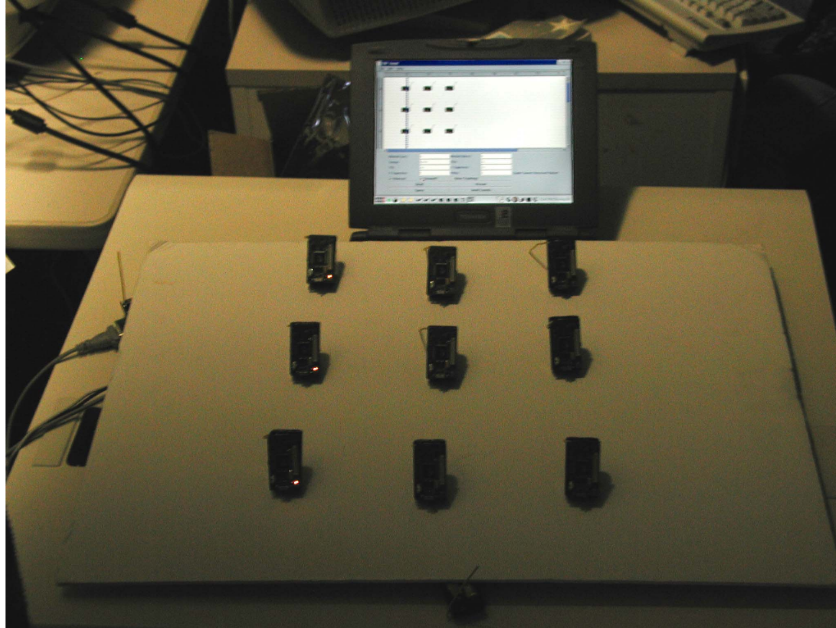
With the obvious limitations of debugging with only 3 LEDs, a protocol was designed to extract state information from the motes. A separate message handler (am_msg_handler_8 wired to tbf_query_event) was created to return this information on demand. Currently only a neighbor list query is implemented. The structure of the protocol is such that it will allow for expansion for the type of queries and the return types. Programs originating the queries can reuse the TBF header structure to send query messages. Query Type is put into the TOS field of the TBF header. Query replies for neighbor lists begin with a 1 byte Moteid of the replier, then 1 byte for the number of neighbors, followed by the Moteids of the neighbors (each 1 byte).

Writing TBF applications

So far we discussed the design and implementation of TBF on the Mica Motes. This corresponds to the networking layer in the network stack. It is evident that this is just the framework and applications need to interact with the network layer in a well-defined manner, also we need to support multiple applications. The process flag in the TBF header is used to decide whether a received packet must be sent up the stack or not. The TOS field is used as a multiplexing key to deliver the message to the appropriate application. Any application that is interested in a particular type of data must register a function that needs to be called when a message of that type is received. We have provided the tbf_tos_register() and tbf_tos_deregister() functions. This design closely matches with the TinyOS philosophy of active messages. The functions are given full access to the TBF headers and they can modify them before they are forwarded any further. In future versions, this will be more tightly integrated with TinyOS so that the signaling mechanism used to trigger events will be made use of, instead of just calling a function.

Demo setup

The first and simplest demonstration of TBF was a 3 x 3 grid of nodes (1 x 3 grids were good initially, but were otherwise unremarkable). Numbered from 0 to 8, the nodes were given their absolute locations and allowed to stabilize their neighbor tables. A node was chosen as the source, and another as the destination. All the TBF constructs were encoded into an RF packet using the TBF Viewer (GUI), which in turn sent the packet to the SerialForwarder, and ultimately sent out through the UART to the waiting base station. The propagation of the packet could be seen with the toggling of the yellow LED on the MICA Motes. A description of each of these components follows.



Forwarding packets along a vertical straight line. The yellow LEDs lit indicate the path the packet followed. The GUI display shows the vertical line representing the trajectory.

Generic base

The generic base is simply a Mote burned with code to forward packets from the UART to the RF and vice versa. Its sole purpose is to act as a communication conduit between the computer and the sensor network. The code is included as part of the TinyOS distribution in nest/apps/generic_base.

Serial forwarder

The serial forwarder exists because many different applications may need to access the serial port for the purposes of communicating with a network of motes. It acts as a traffic-multiplexer for packets going out to the base station and packets coming in from the base station. Applications wishing to use the base station must create a socket connection to the serial forwarder and write/read from the socket. The SerialForwarder is written in Java, but this is irrelevant to the application, as any program capable of creating TCP connections will be able to communicate with the forwarder. The serial forwarder also allows for implementations that put the application and the base station on different machines, as long as there is a network connection between the two. More information about the serial forwarder can be found in the PDF documentation in the TinyOS distribution.

TBF viewer

The TBF Viewer Java GUI interface to TBF allows the user to easily inject TBF packets, visualize network topography, query nodes and more. The user simply has to

input the necessary fields, the parametric trajectories and other TBF constructs. The application receives and parses packets and displays the results on screen. When injecting a TBF packet, the application encodes a TBF packet and sends it to the serial forwarder.

When considering what development environment/language to use for the GUI, Java naturally came to mind. Rapid application development and portability were the main reasons that Java would work well in this situation. There were, however, drawbacks to using Java – including data conversion and speed.

Future work and Conclusion

TBF is practical to implement on extremely resource constrained embedded devices. There are a slew of applications for this new technology, both in civilian and military contexts. The forwarding decision is made locally and in constant time irrespective of the network size, thus being scalable. Some of the issues we did not cover were better encoding of the trajectory to reduce size, including compression techniques. Also, RPN can be slow to evaluate since it is interpreted; many ideas from active networks can be easily applied to speedup this process.

Service discovery with TBF

There exist many publisher/subscribe models for sensor networks; many of them rely on flooding to locate the publisher of data. We believe flooding is expensive and not scalable. The idea is to implement a fully functional service discovery protocol based entirely on TBF. The advantages are many, including longer battery life and lesser network congestion. The general approach we intend to take is as follows: Each publisher of a data sends out the advertisement for the service along radial lines (emanating from its current position) throughout the network. This message is cached within the network. When a node is interested in a particular type of service, it sends out another message along radial lines. By simple, geometry it is evident the two sets lines will intersect at some points in the network. One of the nodes near the intersection will reply back with more information to the subscriber about the location of the publisher. It is easy to see why a naïve implementation will not scale beyond a few hundred nodes, due to memory requirements. Our aim is to design an efficient and reliable protocol for this purpose.

References

[1] The Museum of HP Calculators http://www.hpmuseum.org/rpn.htm
[2] TinyOS http://webs.cs.berkeley.edu/tos/
[3] Trajectory Based Forwarding http://www.cs.rutgers.edu/~dnicules/research/tbf/tbftr.pdf