Interactive Toy Routing: The CyberBeanie Packet Forwarding Protocol.

Andrew Lamb
6.033 Design Project One
3/22/2001
Saltzer/Clarke TH2

Abstract

A protocol is described for routing information across a wireless ad-hoc network of interactive toys. The design presented considers both the severe constraints of the toys' hardware, and the requirements of the intended application. Each CyberBeanie toy only stores a single route through the network in its RAM. The actual messages flowing through the network contain routing instructions as part of their data. Intermediary CyberBeanies determine how to forward packets based on the routing information they contain. Interesting routes are inferred from observing broadcast messages. Each broadcast message contains a route from its source to its current location. Performance is improved by limiting the propagation of broadcast messages. Based on preliminary calculations, the CyberBeanie packet forwarding protocol supports its intended application well, providing a high degree of interactivity.

Introduction

A CyberBeanie is a furry ball about three inches across. The CyberBeanie's key feature is that each one forms a "friendship" with another CyberBeanie. A pair of friendly CyberBeanies reflect each others' simulated "emotions," meaning that if one is heated up in the palm of a hand, an light on the other turns on in response.

Each CyberBeanie has a very fast embedded microprocessor. The memory available to each CyberBeanie is limited to 256 Bytes, and each CyberBeanie has a radio transmitter which can communicate at 1000 bits per second over a distance of four feet. By cleverly using spread spectrum technology, interference is avoided and simultaneous transmissions are correctly received by all CyberBeanies in range.

The manufacturer preprograms a unique 32 bit identification number into each CyberBeanie which serves as a network identifier. Each CyberBeanie generates 16 bit temperature readings which are used by its friend to determine how often to flash its light. If a CyberBeanie is within four feet of its friend, they communicate directly. If there is a path for temperature messages to travel indirectly through intermediary CyberBeanies, the friends can still communicate even if they are more than four feet apart. To act as routers for each other, CyberBeanies come equipped with a network layer packet forwarding protocol.

The most common use of CyberBeanies is estimated to be in a square classroom with 25 students. Hence, the CyberBeanie packet forwarding protocol in this paper was configured for a scenario of 25 CyberBeanies arranged in a grid. CyberBeanies can talk to only their immediate neighbors, and the CyberBeanies on diagonals are not close enough to receive transmissions.

This paper describes the CyberBeanie packet forwarding protocol. A high level description is presented to give the reader an general overview of the design. Then the specific packet formats, and network layer pseudo code are presented with detailed descriptions of their purpose and function. Then, a detailed analysis and justification of design decisions is presented, followed by a conclusion and suggestions on future work.

Design

The CyberBeanie packet forwarding protocol consists of two packet types. First, each CyberBeanie advertises its position in the network to its friend using Find Friend packets. When CyberBeanies receive Find Friend packets not from their friends, they add their own ID to the end of the Find Friend packet, and send out the packet again. Hence, each Find Friend packet contains a history of all CyberBeanies it has encountered, and therefore a possible route through the network back to the sender. Each CyberBeanie also listens for Find Friend packets that originated at its friend. When a packet addressed from its friend arrives, the CyberBeanie saves the route contained in the packet and uses that route to send temperature messages to its friend.

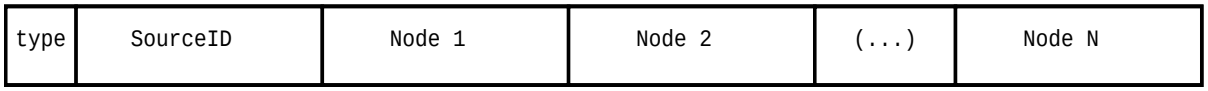
Unchecked, blind forwarding of Find Friend packets would soon clog the available radio bandwidth with multiple copies of the same Find Friend message. Therefore, CyberBeanie's keep a list of recently seen source IDs for Find Friend packets and drop any new packets with the same source ID.

Once a CyberBeanie has a route to its friend, it sends the friend Friend Temperature packets. Each Friend Temperature packet contains all of the routing information necessary to be delivered to its recipient. When a CyberBeanie receives a Friend Temperature which is not addressed to itself, it uses the route information contained in the packet to figure out how to handle the packet. If the current CyberBeanie is the current node on the route it forwards the packet on to the next Cyber-Beanie. If the receiving CyberBeanie is not the next node on the route, it drops the packet. Before resending the packet, the packet's current node is updated. Only the current node on the route will ever transmit the packet, conserving network bandwidth.

Find Friend Packet

Figure 1: Find Friend packet layout

Find Friend



1 32 32 32 32

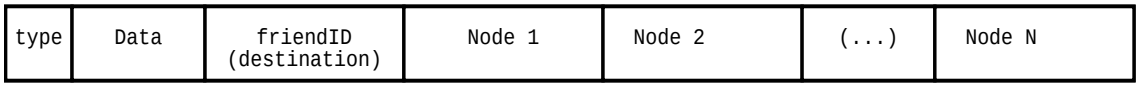
Total size of the Find Friend packet = 1 + 32*(N + 1)
where N = the number of nodes that this packet has been forwarded by)

The Find Friend packet is broadcast when a CyberBeanie needs to advertise its position to its friend. Every station remembers the IDs of the sources of the last seven Find Friend packets it has received. Every time the Find Friend packet is retransmitted, the transmitting node appends its own ID to the packet, causing the packet's length to grow by 32 bits

Friend Temperature Packet

Figure 2: Friend Temperature packet layout

Friend Temerature



1 16 32 32 32 32

Total size of a Friend Temperature packet = 17 + 32*(N + 1)
where N = the number of nodes that this packet must be forwarded through))

The Friend Temperature packet delivers 16 bit temperature data from a CyberBeanie to its friend. The Friend Temperature packets each contain a sequence of IDs that represents a route through the CyberBeanie mesh network. As each node on the path retransmits the Friend Temperature message, the node removes its ID from the packet's path. Hence, the packet's size decreases by the 32 bits.

Global Variables

Figure 3: Pseudo code defining global variables

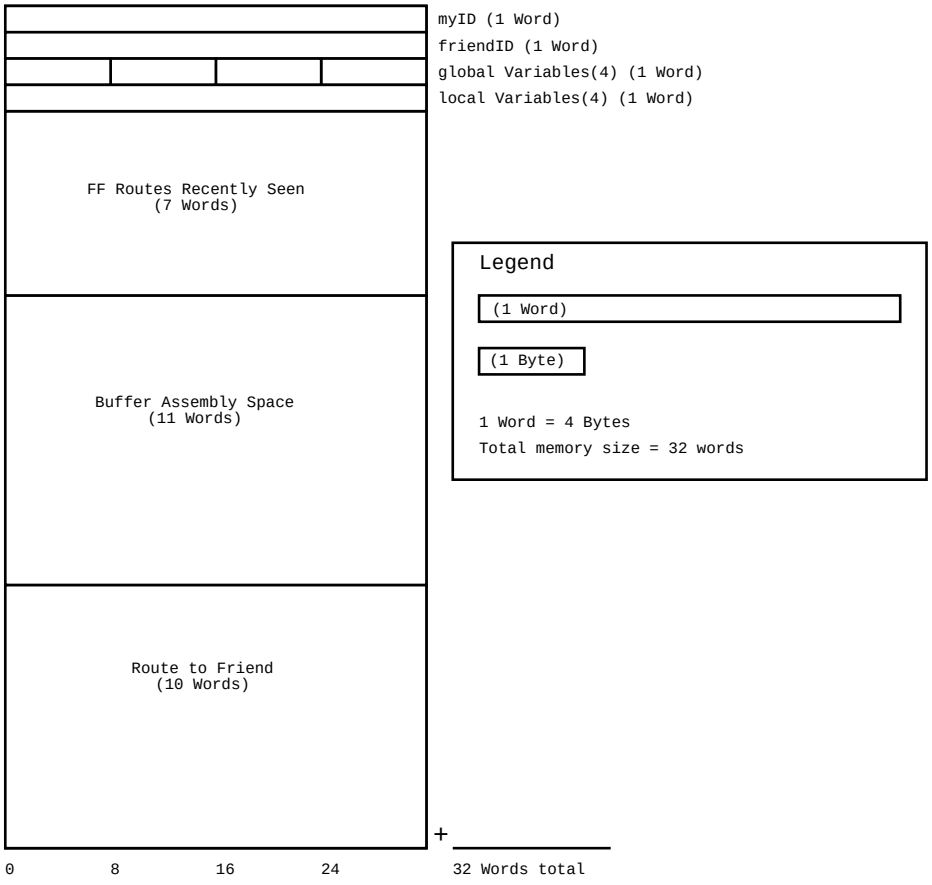
```
// Global variables

Word myID;              // This CyberBeanie's unique 32 bit ID
Word friendID;          // Friend's 32 bit ID
```

```
Byte wait_count;         // the number of attempted temperature messages
                         // without a route before sending out another
                         // Find Friend packet

Byte message_count;      // The number of temperature messages that the
                         // CyberBeanie will send without getting messages from
                         // its friend.



Byte route_length;       // size of route to friend (necessary to copy route)
Byte most_recently_seen; // memory space to store the index of the most
                         // recently seen find friend packet (needed to implement
                         // the most recently seen functions called upon in the
                         // rest of the code

Word recently_seen[7]    // the seven most recently seen find friend ids

Word friend_route[10];   // route to friend

Word buffer[11];         // buffer to assemble network packets in
```

The global variables in Figure 3 represent the memory allocation scheme for the five by five grid of CyberBeanies described in the introduction. These global variables are referenced in the following pseudo code using the prefix `global` to differentiate them from local variables.

Memory Allocation

Figure 4: Memory Allocation for each CyberBeanie



The CyberBeanie contains 256 Bytes of random access memory (RAM) for use by the routing protocol. This memory is logically divided up into 32 32 bit Words. The first Word is reserved for the CyberBeanie's own ID. The second Word is reserved for the CyberBeanie's friend's ID. The third Word is allocated into 4 Bytes which hold the global variables `wait_count`, `message_count`, `route_length`, and `most_recently_seen`. The fourth Word is set aside for local variables and remembering buffer lengths. 7 Words are allocated for IDs on Find Friend packets which have been recently received. 11 Words are allocated for buffer space in which packets are modified. The final 10 Words are allocated for storing a route to the CyberBeanie's friend. The specific sizes of the global variables were determined by the hardware limitations as described in the Discussion section below.

Global Constants

Figure 5: Global constants for network layer

```
// Constants programmed into the CyberBeanie's at the factory

// first bit of each message is the type field
#define TYPE_FRIEND_FIND = 0;
#define TYPE_FRIEND_TEMP = 1;

#define MAX_WAIT_COUNT = 3;    // number of network requests without a
                               // route before we send another FF packet
#define MAX_MESSAGE_COUNT = 3; // number of messages to send without
                               // recieving any messages from friend before
                               // assuming route is bad
#define MAX_PACKET_SIZE = 11;  // that is how much buffer space we have for
                               //assembling packets, so this is
                               //the maximum buffer space allowable

#define MAX_TIME_IN_RECENTLY_SEEN = 1160; // how often we clear the
                                          // recently seen buffer in milliseconds
#define TEMPERATURE_INTERVAL = 776;// the interval, in milliseconds, between
                                   // link_send requests. The larger this number,
                                   // the more Friend Temp messages are sent,
                                   //and the more bandwidth they consume.
```

The global constants in Figure 5 are programmed into each CyberBeanie's read only instruction memory at the factory and can not be changed. The specific values that are assigned to the constants were determined by the radio link speed and the most likely scenario.

Network initialization code

Figure 6: Pseudo code for initialize_network()

```
// Code to initialize the network

void network_initialize(my_id, my_friend_id) {
  // initialize global variables
  globals.myID = my_id;
  globals.friendID = my_friend_id;
  globals.wait_count = 0;
  globals.message_count = 0;
  globals.route_length = 0; // no route to start with
  globals.recently_seen

  // send initial Find Friend packet
  network_sendFF();

  // set up a timer that removes all IDs from the recently seen
  // list of Find Friend packets forwarded. This is done
```

```
  // so that if the network changes configuration after initial
  // stabilization, the CyberBeanies will respond to find friend broadcasts
  set_timer(MAX_TIME_IN_RECENTLY_SEEN) {
    if (globals.recently_seen.updated) {
      globals.recently_seen.clear_list();
    } else {
      globals.recently_seen.updated = false;
    }
}
```

The initialization code for the CyberBeanie packet forwarding protocol initializes the global variables and sends an initial Find Friend message into the network. The initialization code sets a timer that will go off every MAX_TIME_IN_RECENTLY_SEEN milliseconds which will clear the global list of recently seen Find Friend packet source IDs no Find Friend packets have been seen recently.

Network layer transport code

Figure 7: Pseudo code for sending packets

```
// Code to send data over the CyberBeanie network

// temp_data is a 16 bit value specifying the CyberBeanie's
// current temperature that needs to be sent to the
// CyberBeanie's friend

void net_send(temp_data) {
  // if we have a route to our friend, send the temp to the friend
  if (global.route.length > 0) {
    // assemble the packet for transmission
    global.buffer.type = TYPE_FINDFRIEND;
    global.buffer.temperature = temp_data;

    // copy the route to our friend into the message buffer
    global.buffer.route = global.friend_route;

    // send the packet (link_send's length is in number of bits)
    // 1  bit for type
    // 16 bits for temperature data
    // 32 bits for each id on our node
    link_send(global.buffer, 1 + 16 + global.route_length * 32);

    // increment the global message count so we can
    // detect a change in the network configuration
    global.message_count++;

  } else {
    // else, we don't have a route to our friend yet
    // increment the message wait counter so we give the
    // Find Friend packets time to find our friend
    global.wait_count++;

    // if we have waited long enough without hearing from our
    // friend, try resending a Find Friend message
    if (global.wait_count > MAX_WAIT_COUNT) {
      // send a Find Friend message and reset the wait count
      network_sendFF();
      global.wait_count = 0; // reset the timer
    }
  }

  // if we have sent several temperature messages without getting
  // any temperature messages from our friend, assume that the
```

```
  // route we have is no longer valid, so we clear the routing
  // information that we have.
  if (global.message_count > MAX_MESSAGE_COUNT) {
    global.route_length = 0; // clear the route
    global.wait_count = 0;   // reset the message wait count
    network_sendFF();        // send an initial Find Friend message
  }

  return;
}
```

(Note: The mechanics for setting the correct bits in the buffer is not important to the packet forwarding protocol. Memory was allocated for this task, and references such as `globals.buffer.type` refer to the type field of the global buffer.) If the CyberBeanie knows a route to its friend, it sends out a Friend Temperature packet. If the CyberBeanie doesn't know a route to its friend, it checks to see how long ago it last broadcast its own position. If a suitable amount of time has elapsed, the CyberBeanie sends another Find Friend packet assuming that its previous one was lost.

If the network has changed configurations and the CyberBeanie's path to its friend is no longer valid, a new route must be determined. A network configuration change is detected by keeping a count of the number of Friend Temperature packets that the CyberBeanie has sent without receiving any Friend Temperature messages in reply. If the unanswered message count reaches a predetermined threshold, the currently held route is invalidated and the route discovery phase begins again with the broadcast of a Find Friend packet.

Figure 8 shows pseudo code to assemble and broadcast a Find Friend packet.

Figure 8: Pseudo code for sending a Find Friend packet

```
// Code to send a Find Friend packet
void network_sendFF() {
  // assemble a Find Friend packet
  global.buffer.type = TYPE_FRIEND_FIND;

  // append this CyberBeanie's ID so other CyberBeanies
  // know who originally send the message
  global.buffer.source = global.myID;

  // 1 bit for type
  // 32 bits for the one ID
  link_send(global.buffer, 1 + 32 * 1);    // broadcast our whereabouts
}
```

Network layer delivery code

Figure 9: Pseudo code for handling incoming packets

```
// network layer packet handler

void net_deliver(linkBuffer, size) {

  // if we are dealing with a Friend Temp message
  if (linkBuffer.type == TYPE_FRIEND_TEMP) {
    if (linkBuffer.destination = global.myID) {
      // this message is for us, so deliver it to the application layer
      handle_temperature(linkBuffer.temperature);

      // make note of the fact that we got a friend's message by
```

```
      // resetting the unanswered message count to zero thus
      // preventing the route from being discarded
      globals.message_count = 0;

    } else if (linkBuffer.route.currentNode == global.myID) {
      // we are the current node in the route specified by the packet
      // so resend our packet without our node id to the next node
      link_send(linkBuffer, linkBuffer.size - 32); // chop off last ID (ours)

    } else {
      // this packet was not for us, nor are we the current node in the
      // route so discard packet (eg do nothing)
    }

    // Otherwise, we are dealing with a broadcast Find Friend message,
    // which we need to route appropriately
  } else if (linkBuffer.type == TYPE_FRIEND_FIND) {

    // if this packet is from our friend, update out routing information
    if (linkBuffer[1] == global.friendID) {

      // if this is a better route than the one we currently know about
      if ((linkBuffer.size - 1) < global.route_length) {
global.friend_route = linkBuffer.route; // copy route in packet for friend route
global.route_length = linkBuffer.route.length; // update route length
      } else {
// the route advertised is not as good as our current one, so
// do nothing and drop the packet
      }

    } else if (recently_seen.contains(linkBuffer.source)) {
      // else, if we have seen the source of this packet recently, drop packet

    } else if (size < MAX_PACKET_SIZE) {
      // else if our memory allows this packet to be processed
      global.buffer = linkBuffer; // copy the received message(with its route)

      // tack our ID on to the end of the packet and send it on its way
      global.buffer.route += global.myID;
      link_send(global.buffer, linkBuffer.size + 32);

      // make a note that we have seen this particular
      // find friend message so we don't propagate duplicates
      globals.recently_seen.add(linkBuffer.source);

    } else {
      // there was no room to assemble a new packet in our buffer,
      // so we are forced to drop this packet
    }
  }

}
```

If the link buffer contains a Friend Temperature message that is addressed to the receiving CyberBeanie, net_deliver hands the temperature data up to the application layer. The unanswered message counter is then reset. Otherwise, the Friend Temperature was not destined for this CyberBeanie, so the routing information in the packet is examined. If the CyberBeanie's ID is the ID of the current node in the path specified in the packet, then the current ID is removed from the packet's path, and the packet is retransmitted. If the CyberBeanie was not the next node on the path, the packet is ignored.

If the link buffer contains a Find Friend message that is from the CyberBeanie's friend, the packet's route is examined. If the route is shorter than the one that is currently stored, the currently stored route is replaced. If the Find Friend packet is not from the CyberBeanie's friend, then its source is compared to a list of source IDs from recently seen Find Friend packets. If the source of the Find Friend packet has been recently seen, then the packet is discarded. If the source hasn't been recently seen, then the CyberBeanie appends its own ID number to the route and retransmits the packet after adding the packet's source ID to the list of recently seen IDs.

Discussion

The implementation of the CyberBeanie packet forwarding protocol was guided by the available hardware. The most severe limitations on the CyberBeanie packet forwarding protocol are the paucity of available memory and the slow radio link. The CyberBeanie's very fast microprocessor, on the other hand, can be used to offset the disadvantages of the memory and radio links. Furthermore, the specific use of this routing protocol for interaction among toys allows optimizations that would not be permissible for other more general purpose routing protocols.

Design Rationale

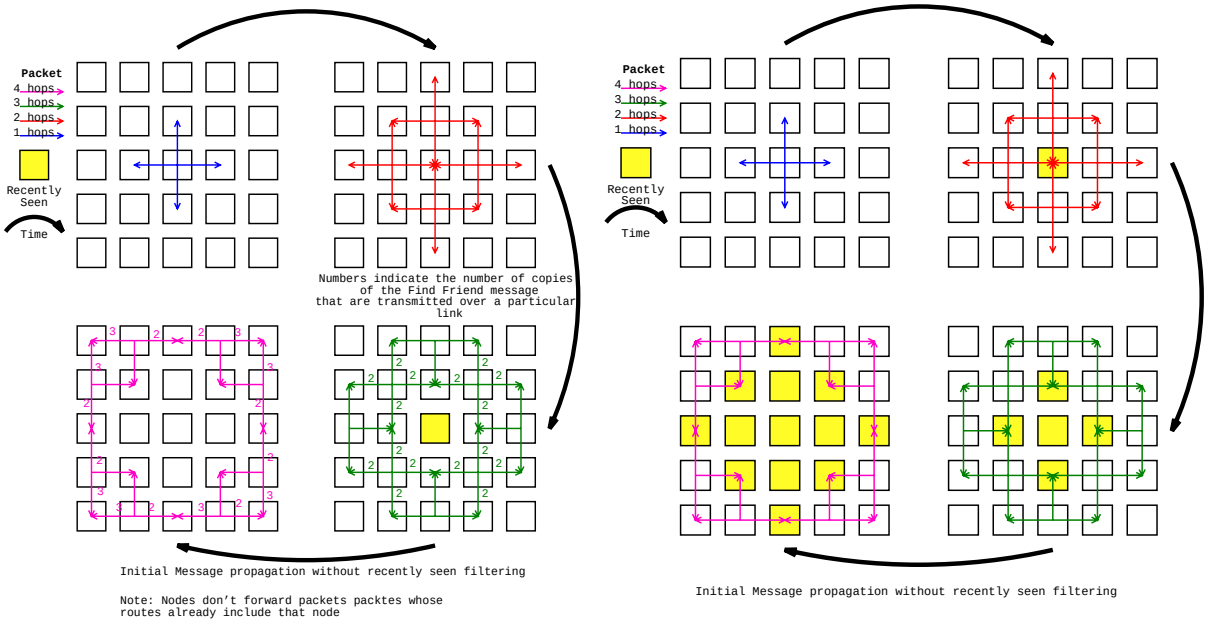
Simplicity was a major design goal. The simpler a system is, typically the better performance it has and the less error prone it is.

Initially, a routing scheme based on routing tables was considered, but was then discarded because of insufficient memory. Instead, the speed of the processor was utilized to process and modify incoming packets before retransmission. Further simplification was obtained because the friendships between CyberBeanies are symmetric. Since the CyberBeanie's end layer interface is a flashing light, it is unnecessary to guarantee every single packet arrives at its destination. It is a natural choice to use any Friend Temperature message originating from a CyberBeanie's friend as an acknowledgement that the CyberBeanie's own packets are getting through. By removing the notion of acknowledgement packets from the CyberBeanie packet forwarding protocol, half of the potential packet traffic was eliminated.

Remembering recently seen Find Friend messages seems opposed the stated goal for simplicity. Keeping a recently seen list adds complexity to the overall routing system, consumes more memory and therefore makes for shorter maximum route lengths. However, the performance benefit realized by not wasting bandwidth with duplicating packets more than justifies the costs of remembering recently seen IDs.

An alternate approach which was considered for reducing packet duplication was discarding Find Friend packets that the receiving CyberBeanie had already seen. Figure 10 shows the performance of the alternate approach on the left. The approach of remembering recently seen CyberBeanies is shown on the right. While the alternate approach does allow the receiving CyberBeanie to choose from more paths through the CyberBeanie mesh, it consumes a huge amount of network bandwidth because Find Friend packets grow in size after each retransmission. The severely limited bandwidth of the available radio link makes removing duplicates worth the additional complexity of remembering source IDs from recently seen Find Friend packets.

Figure 10: Initial Find Friend propagation with and without recently seen



Scalability

The route storage needs to be N Words, where N is the largest allowable number of hops through the network. The buffer space needs to be N Words + 17 bits in order that type, data and the largest permissible route can be created in the buffer to send. The number of recently seen packets also needs to be roughly N (though N/2 would work as well) in order to accommodate the number of expected Unique IDs. The space allocated for global and local variables remains constant at 4 Words. Hence, the amount of memory utilized by this design as a function of the number of CyberBeanies is on the order of 3N.

Memory usage scales well in this design because most of the information necessary for routing a particular packet is stored inside the packet during transmission. The network bandwidth is the limiting factor when scaling this system.

Initialization

The CyberBeanie packet forwarding protocol is said to be stable when all of the CyberBeanies have routes to their friends and are broadcasting temperature data happily. The network speed directly determines the speed at which a mesh of CyberBeanies stabilizes because it determines the speed at which Find Friend packets can propagate across the network. The calculations below are based on the five by five grid scenario described in the introduction. To determine the average setup time of the CyberBeanie network, consider the individual worst case scenario where the two friends are located at opposite corers of the five by five grid

By tracing out the message propagation paths (see Figure 11), the maximum route length is 8 nodes, with a propagation time of approximately 1160 mS. The total number of Find Friend packets caused by a particular initial broadcast is the same regardless of the two friend's relative locations. Most Find Friend packets will not start at the edges, and hence their retransmissions will not

consume as much bandwidth because fewer nodes will be visited. Since the hop count is smaller, the average packet size which grows linearly with the number of hops will be smaller, and the total network bandwidth consumed will also be smaller.

12302 bits of the total bandwidth of the network are required for all packets in the worst case scenario presented in Figure 11. The total network bandwidth equals the total number of bits that can be sent per second over the entire network. In the five by five grid scenario, the total network bandwidth is

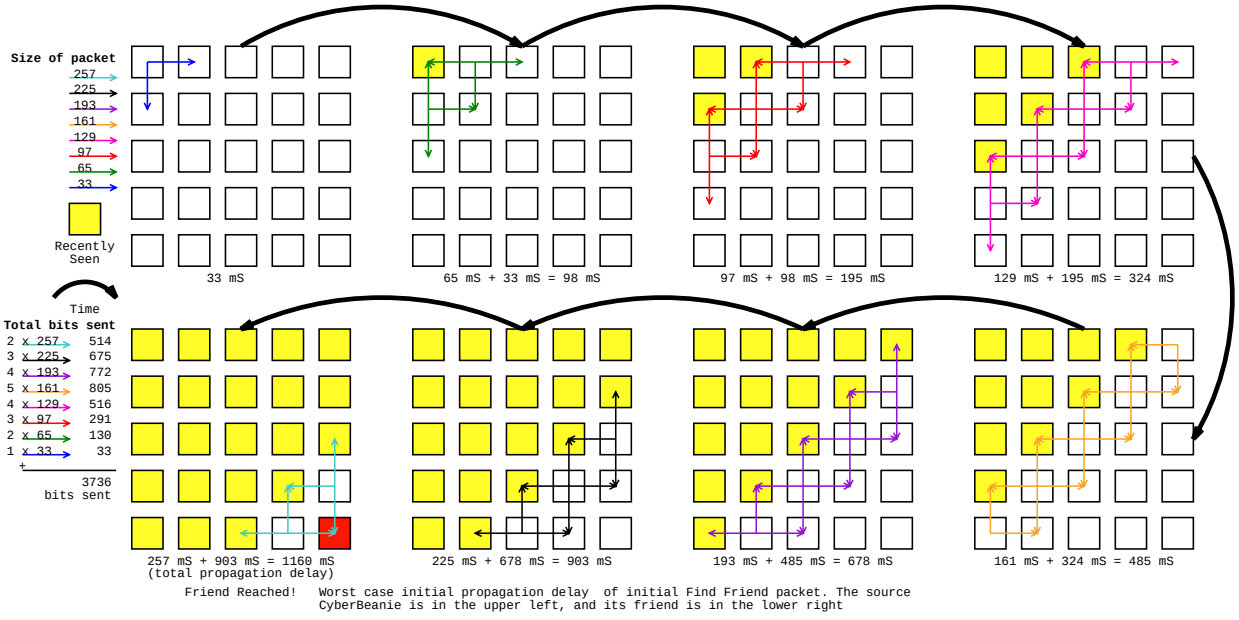
25 CyberBeanies x 1000 bits/sec/CyberBeanie = 25,000 bits/sec

Using the total network bandwidth and the worst case cost of sending a Find Friend message, the average time taken to initialize the network is approximately

25 CyberBeanies x 3736 bits/initialization/CyberBeanie x 1/25,000 secs/bit = 3.736 sec

This is only approximate because it does not account for potential bottle necks at the central CyberBeanies. On the other hand, since it assumes the worst case, the approximations will (hopefully) partially offset each other.

Figure 11: Worst case initial Find Friend message propagation



Interactivity

The faster each CyberBeanie sends its friend temperature data, the more interactive the toy will be. The physical constraints of the network, impose limitations on the rate at which data can be exchanged. By examining the worst case Friend Temperature message propagation, the exchange rate for the five by five grid can be computed.

Figure 12 illustrates that the worst case Friend Temperature message takes 1288 milliseconds to reach the specified friend, at an expense of 1288 bits of total network bandwidth. Using the worst case message scenario the network will support

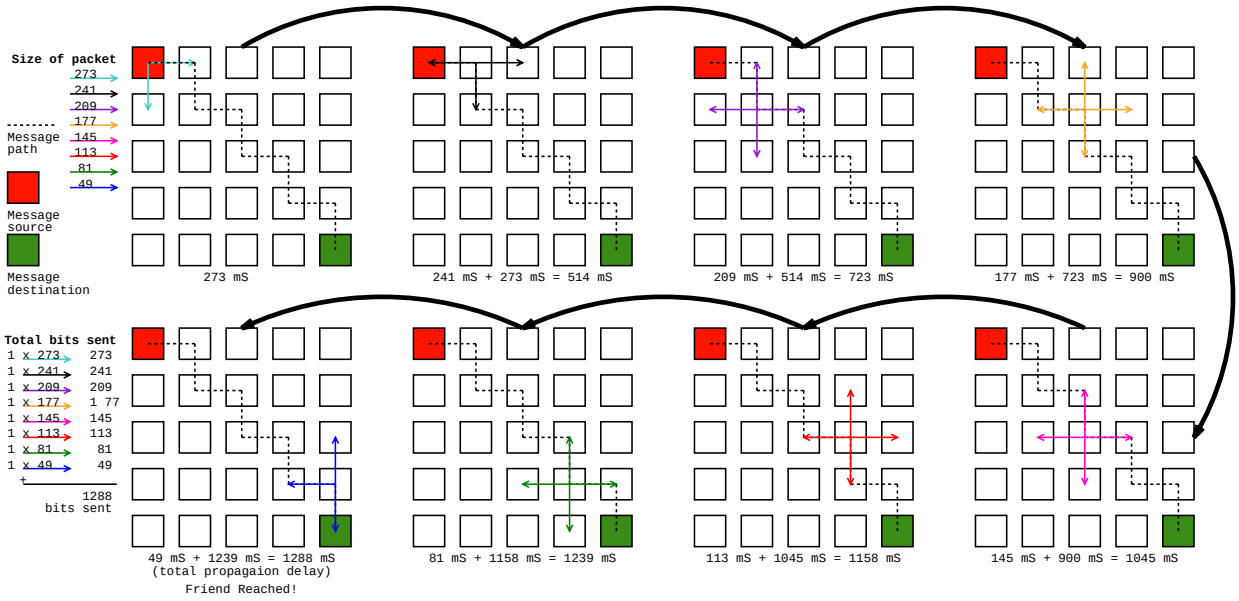
1/1,288 message/bits x 25,000 bits/sec = 19.4 messages/sec

implying that each CyberBeanie can send out Friend Temperature messages every

1/25 1/CyberBeanie messages x 19.4 messages/sec = 0.776 sec = 776 mS

These calculations do not take into account the potential bottle necks of the CyberBeanies in the middle of the mesh. CyberBeanies route messages along the shortest known path without any regard for the current traffic load of the center CyberBeanies. The amount of bandwidth consumed by a particular Friend Temperature message is proportional to the number of hops necessary to reach the friend. It is unlikely that all paths will go through the middle CyberBeanies, but it is also unlikely that all paths will be the worst case. The worst case bandwidth and the potential bottlenecks partially cancel each other out.

Figure 12: Worst Case Friend Temperature message propagation



Support for intended application

The algorithms of the CyberBeanie packet forwarding protocol are general, and are suitable for configurations other than the five by five grid. In the setting of 25 school children sitting in neat rows, the performance of the network is good. Each CyberBeanie can expect to be informed of its friend's temperature every 776 milliseconds. To the end user, a 776 millisecond response time will make the CyberBeanies very interactive. The initial setup time of 3736 mS, while noticeable, is reasonable.

The CyberBeanie protocol will take 4393 milliseconds to react to changes in the network. It takes 3 * 776 milliseconds to notice that no friend messages have arrived, and then 1288 additional milliseconds to receive a new Find Friend packet from its it friend. It then takes 776 milliseconds longer for the first Friend Temperature to arrive.

If less stringent requirements were placed on system, such as only 100 unique ID numbers (necessitating only 7 bits to store an ID, and an allowable 20 second stabilization time, both memory and radio link bandwidth could be decreased. Memory consumption of the CyberBeanie packet forwarding protocol is directly proportional to the size of the unique IDs required, and hence reducing the ID size by a factor of four would reduce the memory consumption by a factor of four. Because IDs make up the largest portion of transmitted packets, a factor of four reduction in ID size would result in a factor of four decrease in the necessary speed of the radio links as well.

Factory Programmed Constants

To determine the global constants for WAIT_COUNT, MESSAGE_COUNT, and MAX_TIME_IN_RECENTLY_SEEN in the five by five grid, the worst case propagation times were used. Using the worst case makes the CyberBeanies slightly less interactive as it takes them longer to notice a change in network topology and it potentially under utilizes the available radio link's bandwidth. However, using worst case times makes the CyberBeanies more robust to deviations from the five by five grid topology.

WAIT_COUNT determines the number of times a friendless CyberBeanie attempts to send a temperature before it launches another Find Friend packet. Propagating a Find Friend message through the CyberBeanie mesh network consumes a large amount of bandwidth, so not sending redundant messages boosts performance. A CyberBeanie should wait at least the worst case propagation time(1160 milliseconds) of a Find Friend packet before sending another one. To allow for network congestion, and to make absolutely sure that the Find Friend message has propagated through the network, WAIT_COUNT is set to 3 inter temperature times for a total of 3 x 776 mS = 2328 mS between Find Friend packets.

MESSAGE_COUNT is the number of temperature packets that are sent to a friend without receiving acknowledging temperature packets. When MESSAGE_COUNT packets have been sent, the current route is invalidated. If MESSAGE_COUNT is set too low, a few lost packets might cause the CyberBeanie to discard good route information and rebroadcast a costly Find Friend packet. If MESSAGE_COUNT is set too high, a CyberBeanie's interactivity is decreased because it takes longer to realize that the network configuration has changed. Allowing 3 full worst case message propagation times(2 x 1,288mS = 2,576mS) strikes a compromise between these two trade-offs.

MAX_TIME_IN_RECENTLY_SEEN is the number of milliseconds of not seeing any Find Friend packets before a CyberBeanie discards its table of recently seen Find Friend IDs. A good choice is the time that a worst case Find Friend packet takes to propagate through the network (1160 mS). If no Find Friend packets have not been observed in this time, there is a good chance that the network topology has stabilized.

Conclusion

The CyberBeanie packet forwarding protocol described in this paper offers good support for its intended application. As with all designs, the CyberBeanie packet forwarding protocol does have limitations. The most severe limitation is the maximum hop count. It would be wonderful if an arbitrary number of hops was allowed, which is a promising area of future research. An additional limitation is potential bottlenecks within the system. If all of the shortest paths in a CyberBeanie mesh network happen to go through one particular CyberBeanie, the analysis presented in this paper is totally invalidated. Without any data about how the CyberBeanie friendships are formed or their spatial distribution, considerations about bottlenecks could not be included in the calculations. The linear scaling of resources used by the CyberBeanie packet forwarding protocol is not ideal, but it is not unreasonable given the severe limitations imposed by the hardware. The linear increase in network bandwidth as the maximum hop count grows is much more worrying. As distributed mesh networks become increasingly more common, protocols to effectively route packets among peers without linearly scaling bandwidth requirements will become increasingly more important.

Bibliography

Jerome H. Saltzer and M. Frans Kaashoek, Topics in the Engineering of Computer Systems (working title). Cambridge, MA: MIT Laboratory for Computer Science, 2001.

I spoke briefly with both Kailas Narendran and Winnie Yang about this design project