APPLICATION NOTE

Working with Large Networks

Version 1.0

Wireless Technology to Control and Monitor Anything from Anywhere™

500 Discovery Drive
Huntsville, Alabama 35806
877-982-7888

Doc# 600044-01A

Table of Contents

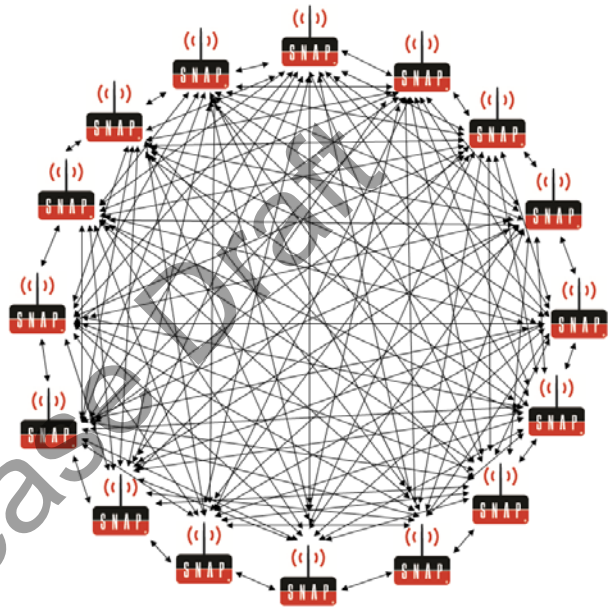
Revision History

Previous Version	Change	Page

The Problem

The SNAP Network Operating System has been optimized to work with the size network that most typical installations are likely require. Its automatic mesh networking provides tremendous power and versatility for how networks can be configured. But some installations require very dense node installations, where every node might be within radio range of dozens of other nodes. In this environment as each node generates the communications necessary to maintain the mesh infrastructure, there can be troubles with too many nodes trying to talk at the same time, blocking each other's communications.

In smaller installations this can be alleviated somewhat through the use of the Carrier Sense and Collision Detect abilities built into SNAP. But waiting for your turn to talk on a clear channel when there are dozens or hundreds of other nodes wanting to talk too can seriously affect throughput performance.



Problem Details

SNAP networking uses two types of communications: multicasts, and unicasts.

A multicast message goes out and is acted on by any node that hears it (subject to multicast group membership[1]). Multicast messages are sent with a "time to live" (TTL) value that indicates the number of times the message will be forwarded by other nodes when a new node hears the message.

Unicast messages, on the other hand, are addressed to be acted upon by a single node, though the message may pass through other nodes if necessary in order to reach its destination.

Why Multicast Matters

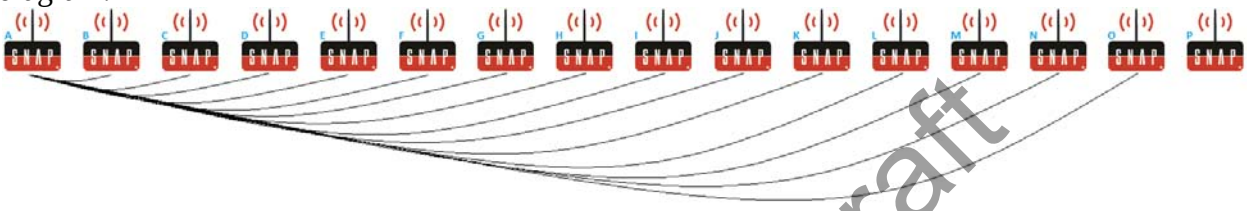
Any time a node hears a multicast message with a TTL greater than 1, it will automatically forward the message on to every other node within range before acting on the message itself. This can cause issues as the forwarding nodes talk over each other in an attempt to pass the message along.

If you have a network with 16 nodes that are all within radio range of each other, and one node sends a multicast message with a TTL of 2 or more, each of the other 15 nodes will do their best

[1] Multicast groups can be used to fine tune your multicast interactions. There are sixteen multicast groups available, and any given node can be associated with any number of these groups – as a node that will act on messages addressed to that group, as a node that forwards messages to that group on to other nodes (if there is any TTL remaining), or as a node that both acts on and forwards the messages. Unless specifically stated otherwise, the examples described here assume that the nodes discussed are set to both act on and forward the messages described. See the SNAP Primer for examples of how the multicast group can be used to your advantage.

to be helpful and attempt to forward that message – all at the same time. If these are the only nodes in the network, and all are in range of each other, this will not be an issue as there are no other nodes that will need to hear the message, so there is nothing to be lost if some of the 15 transmissions are lost because of everybody talking over each other. (In this situation, it would be more sensible to originate the message with a TTL of 1 so none of the receiving nodes would attempt to automatically forward it in the mesh.)

But consider a topography where you have a series of nodes in a row, and nodes at one end of the line are not within radio range of nodes at the other end of the row. In a worst-case scenario, a node on one end of the range might be able to reach all except the very farthest node, as in this diagram.



With this topography, it would be necessary to set the multicast TTL to at least 2 so the message would be forwarded to the node farthest from the message source. But since 14 of the 16 nodes hear the initial broadcast, all 14 of them would (by default) attempt to forward the message, likely obscuring each other's transmissions some in the process. The last node in this situation is likely to still get the message, but there are more opportunities for the message being lost, especially as the size of the network increases and the number of hops necessary to reach the farthest nodes increases.

The SNAP Primer document addresses this issue and proposes some configurations that can reduce the problems of large networks, but there is no single solution that solves every problem. And because time is money, and fine-tuning the configuration of a large network can take some time, standard SNAP mesh multicasting might not be the quickest way to communicate across a large, dense network of nodes.

Can Unicast Save Us?

If multicasts are going to cause problems, can unicasts, or directly addressed RPC calls, eliminate the issues with all these nodes talking over each other? Unfortunately, no.

In order for a unicast message to get from its source to its destination, the source node has to know which "path" of nodes it can use to reach the destination. In the diagram above, node A might be able to use node H as a go-between to get a message to node P. But node A wouldn't know that until it asked, by means of a route request.

A route request is issued any time a node needs to learn how it can reach another node, and in the above example it would come in the form of a question from A, asking "Does anybody know how I can get to 'P'?" Every node that heard the question and that is participating in mesh routing (which, by default, is all of them) would then have to ask the same question, if they didn't already know where P was. Eventually P responds to somebody with "I'm right here!" and the node that gets that response sends a message back to A indicating that A can use it to pass messages to P.

While all of this seems simple enough, the catch here is that all these route request communications are handled by multicast, and as we've already seen that can cause problems in a dense network.

The Solution

You can work around the default mesh infrastructure by implementing a polling framework that uses an acknowledged one-hop multicast structure to determine which nodes are available, and to retrieve information from the nodes.

We will provide sample code for such a framework a little later. But first we will describe the paradigm behind the approach and step through some examples of its use.

How this works

We've already seen that using multicasts in a very dense mesh network can be difficult. But the examples above assume that all the multicast processing (such as forwarding) is handled by SNAP's default mesh networking infrastructure, based on the message's TTL. By sending multicasts with a TTL of 1, and having the script control whether (and when) it forwards the messages, you can minimize or eliminate the issues with nodes stepping on each other's broadcasts.

Once you have established a framework to use single-hop multicasts, your controlling application (Portal, or some application connecting through SNAP Connect) uses the framework functions to discover and query other nodes, rather than the built-in ping, rpc, and mcastRpc functions.[2] Note that this is a query-response arrangement. If you use this framework, your network of nodes cannot "volunteer" information, but can only reply to information requests.

Why it Makes a Difference

What difference should it make if you use a "standard" multicast with multiple hops specified, or if you use multicasts defined within some polling framework? The difference is in the timing.

With a standard multiple-hop multicast, each node that hears the message immediately rebroadcasts the message (with a reduced TTL) before acting on it. When you establish your own framework you can control the timing of the forwarding of the message to keep all the nodes that hear it from talking over each other.

In the sample code below, all messages are sent using multicasts with a TTL of 1, with individual nodes forwarding those messages (also with a TTL of 1) only when their 100 ms timer event fires. Since no two nodes in any group are likely to have their internal timers precisely synchronized with each other (and with the timers' inherent tendency to drift differently over time), this reduces the liklihood that two nodes will step on each other's transmissions.

Sequence of Events: Querying the Network

[2] You still use the mcastRpc function, but you use it to call the framework functions with a TTL of 1 rather than invoking your functions in a more traditional manner.

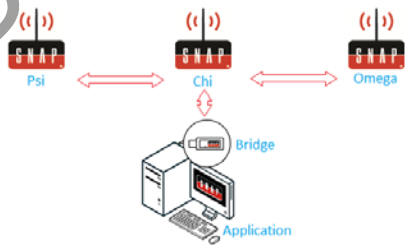
For purposes of keeping things easy to understand, consider how the polling framework would work in a small environment, where you have a PC application, such as a Python script running in Portal, or an application using SNAP Connect; a bridge node connected to the PC, and three other nodes, named Chi, Psi, and Omega, distributed such that the bridge node is within radio range of Chi, but cannot directly reach Psi or Omega, and that Chi can reach both Psi and Omega, though they cannot reach each other.

We'll also begin with the assumption that the application running on the PC does not know which nodes are available beyond the bridge, or how many nodes are out there. Determining this, then, is likely to be the first step in network discovery.

The application's first ping()

The application begins this by sending out a ping() command, as a multicast with a TTL of 1.[3] For the sake of easily referring to things later in this discussion, we will call this command 1. This TTL setting means that the command will only be heard by nodes that are "one hop" from the application, which in this setup means only the Bridge node will hear it.

The Bridge hears the ping()



The Bridge node, on hearing this, ping() command, knows that it is the only node to have heard the command so it knows it can communicate without the liklihood of interfering with other nodes' communications. It immediately constructs a new ping() multicast, also with a TTL of 1 (command 2). Note that in this framework, the SNAPpy script itself is what causes the ping() command to be forwarded, rather than relying on the meth networking infrastructure to forward a multicast command.

Chi hears the ping()

The Chi node hears the ping() sent by the Bridge node, and while it would like to respond immediately, it has no idea how many other nodes may have also heard the command. Being a team player, Chi queues up a route_ping() command to go out the next time its 100 ms timer hook fires. (If any other nodes had heard the Bridge node's ping(), they would do the same thing; the fact that the nodes' timers are unlikely to be synchronized with each other means the nodes' communications would be less likely to interfere with each other.) Once Chi's timer hook fires, Chi sends the route_ping() message as a multicast with a TTL of 1 (command 3). The route_ping() message includes Chi's SNAP network address as a parameter.

Chi's route_ping() message is heard by the Bridge node, and by Psi and Omega. We'll step through how each of these nodes reacts to the message in sequence.

[3] The function names used in this simple description match the names of the functions in the sample code, which we step through a little later. If you were to implement your own framework, you could assign any name of your choice to these functions.

The Bridge responds to Chi's route_ping()

When the Bridge node hears Chi's route_ping() message, it immediately sends an rpA() message (command 4), which acknowledges to Chi that it has heard his route_ping. The rpA() message is also sent as a multicast with a TTL of 1, so it does not generate a wave of network traffic. This acknowledgment lets Chi know that it has been heard, so it will not attempt any retries of its route_ping().

Next the bridge immediately sends a tell_ping() message to the PC application (command 5), reporting Chi's network address so the PC application can keep track of all the nodes it has ever heard from. Because the Bridge node is communicating over a serial cable (or USB connection) with the PC application, there is no concern that the communication will affect over-the-air network traffic. The Bridge node is then done processing this route_ping() message.

Psi and Omega respond to Chi's route_ping()

Meanwhile, Psi and Omega heard the route_ping() message at the same time the Bridge node did. Each of them behaves the same way Chi did when it heard the Bridge node's ping() message – Psi and Omega both enqueue a route_ping() command (each with its own SNAP network address) to go out when each node's 100 ms timer hook fires. It is important to note that neither node sends *any* message immediately upon hearing the tell_ping() message.

Now, let's say that Psi's 100 ms timer hook fires before Omega's does (though it could certainly be the other way around). When Psi's hook fires, Psi will send its own route_ping() message, including its address (command 6). In our configuration, the only node that can hear Psi is Chi. On hearing Psi's route_ping() message, Chi will *immediately* reply with an rpA() route ping acknowledgment message (command 7) so Psi will know it has been heard. Chi will then enqueue its own route_ping() message, this time with Psi's address, to send the next time its timer hook fires.

Meanwhile, Omega's timer hook fires, so Omega sends its route_ping() message by multicast with a TTL of 1 (command 8). On hearing the message, Chi immediately replies with an rpA() (command 9), and then adds Omega's network address to the end of Psi's network address. Psi and Omega are now done in this conversation.

Chi forwards addresses from Chi and Omega

When Chi's timer hook next fires, it will send its queued route_ping() message, this time with the network addresses of Psi and Omega instead of its own address (command 10). The Bridge node, on hearing the route_ping() message, immediately sends an rpA() message (command 11), and the immediately sends a tell_ping() message (command 12) to the PC application to report Psi and Omega's network addresses. This series of commands is now complete, and all radios fall back into silence until some other event or message comes along.

All 12 of these messages are sent as multicast messages with TTL of 1, so none of the messages is automatically forwarded by the SNAP mesh networking infrastructure.

The importance of sequence numbers

If you've been paying attention, you may be wondering why command 3, a route_ping() sent by Chi, causes Psi and Omega to respond by scheduling their own route_ping() messages (commands 6 and 8 in the sequence), while command 10, another route_ping() sent by Chi, does not. This is because all of these messages going back and forth, from the very first ping() sent by the PC application to the final tell_ping() sent to the PC application, contain a sequence number as one of the parameters. Each of the nodes keeps track of the sequence number of the most recent chain of messages it has been involved in so it will know whether it should reply or whether it has already replied to the message.

So when Chi sends out command 3, that will be the first time that Psi and Omega see a route_ping() request with the specified sequence number. They (eventually, when their timer hooks fire) send their own route_ping() messages with the same sequence number. Chi knows it hasn't heard that message from those nodes with that sequence number yet, so it queues their addresses to include in its next route_ping() message (command 10). When that message goes out from Chi, Psi and Omega know they've already done their part for that sequence number, so they remain silent.

The significance of this is, the PC application you have running must carefully keep track of the sequence numbers used for its communications so that any new communication that goes out isn't interpreted by the nodes in the rest of the network as a message to which they've already responded. This is true for ping() messages used for network discovery, and for data requests made by the application.

Sequence of Events: Querying for Data

This leads nicely to the next topic: querying the network nodes for data. Performing network discovery is only half the battle. It is important to be able to determine which nodes are in your network, but if you cannot recover information from them, the knowledge is useless.

In typical network arrangements, once you know a node's address you can make a direct RPC call to the node when you want to query it for data. But as we've already discovered, the multicast route requests necessary to perform RPC calls can be problematic in a large, dense network. So within this polling framework, queries for data are carried out much the same way the network discovery tasks are: with carefully controlled and sequenced multicasts with the TTL set to 1. However just as with an RPC, a query for data must include the address of the node being queried. So in many ways the framework's get_data() queries respond much the same way an RPC might in a smaller network.

The PC requests data

The data request begins with the PC application deciding that it needs some piece of data that only node Omega has (such as a sensor reading). The PC application sends a get_data() message, including Omega's address, as a multicast message with a TTL of 1. Because the bridge knows it is the only node that can hear this request, it knows it can immediately retransmit it, and does so (again with a TTL of 1).

Chi forwards the request

Node Chi hears the get_data() message sent by the Bridge node. Chi has no way of knowing whether any other nodes have heard the message, so it cannot know whether it can forward the message immediately without interference. So, just as with route_ping() messages, Chi queues the message until its 100 ms timer hook fires. At that point, it sends the message, still including Omega's address.

Psi and Omega respond

When Psi and Omega hear the get_data() message sent by Chi, they respond differently.

Psi hears the message and, just as Chi did, prepares to forward the message (as a multicast with a TTL of 1) the next time its 100 ms timer hook fires. It is possible this message will never go out, though.

When Omega hears Chi's get_data() message, it recognizes that it is the message's intended destination and immediately performs the function requested by the PC application. When it gets the result of that function it immediately sends out a get_data_ack() acknowledgment message that includes the return value from the requested function.

The results head back upstream

Chi hears Omega's get_data_ack() message and queues its own get_data_ack() message, to send on its timer hook. When that message goes out, it is heard by Psi and the Bridge node.

From this, Psi knows that it does not need to play any further part in requesting the data, so any get_data() messages it may have had queued will be discarded, and it will enqueue a get_data_ack() message instead.

In our example, Psi's get_data_ack() message won't serve any purpose, because the Bridge node heard the same get_data_ack() message from Chi that Psi did. When the Bridge node hears this, it immediately forwards the data to the PC application using the get_data_result() function. Again, it knows it can send this message immediately because it knows it has a serial connection to the PC application, which will not be a source of interference with any node's radio communications.

Significant Details

We've already seen that it's necessary for the PC application to pay attention to the sequence numbers it uses for its requests, to keep things moving along. But there are a few other infrastructure details that also warrant attention in this implementation.

The Bridge stands alone

You may have noticed that the Bridge node, in the above example, behaves differently from how other nodes behave when it hears certain messages. Non-bridge nodes, when they hear a route_ping() message, repeat the message on their next timer hook, while the Bridge node

instead sends a tell_ping() message to the PC application. The Bridge node also sends a get_data_result() message when it hears a get_data_ack(), rather than just forwarding the get_data_ack().

In order for this to work, the Bridge node has to know that it is functioning as a bridge. The way the script accomplishes this is by checking the "Device Type" NV parameter, parameter 10, in the node when it starts up. If that NV parameter holds the string "Bridge" the node is determined to be the bridge node and will respond appropriately.

If this parameter is not set correctly in the your bridge, this implementation of a polling framework will not function correctly. Similarly, if you have "Bridge" stored in NV parameter 10 in any other node, that node will not behave properly and may disrupt the rest of your network's communications.

Group assignments

Another bit of setup required to make this framework behave as expected is the establishment of all your nodes in appropriate multicast groups. NV parameter 5 is used to assign each node to some combination of 16 multicast groups to which they will respond. The integer stored in this parameter is a bitmap, with each bit representing one of the 16 groups. The pf_setup() function in this example script, which is hooked to run when the node starts up, automatically assigns all nodes to group 257 (0x0101, or 0000,0001,0000,0001b), so the nodes remain in SNAP's default group of 1, but are also included in group 256 (0x0100), which is the group used for all of the framework's multicasts.

If you build your own polling framework, or if you adapt this one, keep group membership in mind. As long as your bridge is a member of all the groups you have in use, you can use this to subdivide your network to even further reduce communications overload. (See the SNAP Primer for examples of using groups to reduce network traffic.)

Outgoing parameters

The get_data() function called by the PC application in this example does not provide any parameters to the end function being called in the node. However this framework provides for sending a parameter along with the data request, using the get_data1() function.

Both these functions end up using the get_data_common() function to actually send their messages out to the network, and if you were to monitor radio traffic you would see that the get_data_common() messages are the ones being sent, rather than get_data() or get_data1(). If you need the ability to send more than one parameter in your data queries, you could adapt the get_data_common() function to handle as many parameters as you need. (This would require that you adapt the existing get_data() and get_data1() functions to match your new get_data_common() function signature, and that you create a new get_dataX() function (where X represents the number of parameters you are sending), or something similar, to pass in the correct number of parameters for you.

Called function names

Calls to one of the get_data() functions (or the inclusion of a data function in a ping command – more on this later) require the name of the function being called on the target node as one of the parameters. This parameter gets stored in the nodes along the way as a string value.

Because SNAP has limited memory available in its nodes, it has a limited number of strings it can keep track of at a time. It has two string pools it maintains: one pool of "tiny" strings, and one pool of "medium" strings. Typically there are more tiny strings available in memory than there are medium strings, but tiny strings can only hold small values, typically eight characters on most platforms. (Medium strings can be at least 62 bytes on all platforms, though some platforms allow for longer strings. If you run out of tiny strings and there are medium strings available, SNAP will automatically use the larger resource.)

If you can keep the names of the functions you would call through the framework at or shorter than eight characters, it would mean that the nodes that are passing your messages along store that value in a tiny string, leaving more medium strings available for other processing. Generally this won't make a significant difference in your scripts. But if you have much processing going on in your nodes between radio transmissions, this is one way you can recover a medium string at the price of a tiny string.

Extended return values

The example walk-through presented earlier has the Omega node returning data based on a get_data() message (actually a get_data_common() message) received. There may be times when your PC application needs to get more than one bit of information back from a node that it queries, either because there are separate pieces of data needed, or because the data it needs to get is too large to fit into a single return message packet.

For the sake of discussion, let's return to the network topography used in the walk-through, where a Bridge node connects to node Chi, which connects to both Psi and Omega. Let us assume the PC application sends a message for the Omega node to run a check_sensors() function on the node, and that the check_sensors() function must be able to return three values.

The check_sensors() function could be written this way:

```
def check_sensors():
    global str_cnt  # Initialized to zero outside the Polling Framework

    str_cnt += 1

    if str_cnt == 1:
        return check_sensor_one()
    elif str_cnt == 2:
        return check_sensor_two()
    else:
        str_cnt = 0
        return check_sensor_three()
```

This way, the first time Chi sends a get_data_common() message to Omega, Omega will run its check_sensor_one() function and return that value. The next time it will run check_sensor_two()

and return that, and the third time it will run check_sensor_three() and return that value. Subsequent requests will repeat this cycle.

You *could* handle this by making three separate requests to check_sensors() from your PC application, or by making a check_sensor_one() request, a check_sensor_two() request, and a check_sensor_three() request. But the latency of the framework communications, especially when trying to reach a node several hops away, could make this collection of request-response transactions take longer than you prefer.

This framework provides an alternate means to return multiple data packets without having to make multiple queries. Instead, start by defining your check_sensors() function like this:

```
def check_sensors():
    global get_data_more  # Declared in the Polling Framework
    global str_cnt  # Initialized to zero outside the Polling Framework

    str_cnt += 1

    if str_cnt == 1:
        get_data_more = True
        return check_sensor_one()
    elif str_cnt == 2:
        get_data_more = True
        return check_sensor_two()
    else:
        get_data_more = False
        str_cnt = 0
        return check_sensor_three()
```


This addition of the get_data_more global variable, declared (and initialized as False) in the Framework code, means that the Framework code will know that Omega has more information to report, and Chi will continue to send requests for more data every 100 ms until Omega returns a get_data_ack() message indicating False for get_data_more. Chi will forward Omega's responses as they are available, meaning the PC application will not have to wait for the first data to make it all the way back before the second data enters the pipeline, and Chi will not have to wait for the PC application to tell it to ask for more information before making the next request. This can significantly reduce latency in a multi-hop network topography.

A side effect of this is that Omega ends up returning messages with sequence numbers beyond the number specified in the original request from the PC application. This is necessary so that the nodes that pass this information on do not assume that messages beyond the first are repeats of the first message and ignore them as already having been forwarded. This means your PC application needs to pay attention to how many responses it expects from such a request, and for its next request (to the same node or to a different node) adjusts its sequence number to account for the numbers "consumed" by the last call.

Retries

Even in a small network like the one used in our examples so far, it is possible for radio communications to be missed due to outside interference or other issues. When you add dozens (or hundreds) of other nodes to the mix, even a carefully planned framework will have moments

where multiple nodes are talking at the same time. It is important, then, to make sure all your communications get through, and to retry any time your message isn't acknowledged.

This example framework allows for retries of route_ping() messages if the node hasn't heard any acknowledgment after 300 ms. It will continue sending the route_ping() message every 500 ms until it reaches the MAX_TTL threshold, in an attempt to ensure that the message gets through.

This is important not only in environments where communications may interfere with each other, but in network where there are so many nodes that SNAP packet size restrictions prevent all the nodes from being able to report in together.

Consider a situation where a Bridge node can reach another node that we'll call Alpha, and that this Alpha node is within radio range of dozens of other nodes. When Alpha hears the ping() message from the Bridge node, it sends the route_ping() message on its next timer cycle, reporting its own address back to the Bridge. This route_ping() message is heard by all the other nodes in the network, who then add their own addresses to the Alpha node's address and, on their timer cycles, send route_ping() messages of their own.

The Alpha node will hear these route_ping() messages, and will start compiling a list of addresses of nodes from which it has heard and sending rpA() acknowledgment messages back to these nodes. Eventually the Alpha node will have collected enough addresses to fill the return packet, and it will be forced to stop collecting new addresses until it can send the route_ping() message back for the Bridge node to hear. Any nodes that have attempted to report in to the Alpha node after the Alpha node ran out of space will not receive an acknowledgment, and will fall into the cycle of retries. As the Alpha node has more space, it will continue to collect these addresses, acknowledging nodes as it does, until all the nodes have been acknowledged (or until they run out of retries).

Patience is a virtue

This pattern of waiting for timer hooks, followed by retry delays of 300 or 500 milliseconds, means that an attempt to query your network, either for discovery or for a data result, can take a bit longer than it would if all the messages were immediately forwarded (and none were lost to communications interference). Even without retries, a network that has five hops worth of communications will take, on average, more than half a second to get responses from "leaf" nodes in your network, and can take as much as a little more than a second.

It is important, then, to not implement time-outs in your PC application that are too short to accommodate the network you are working with. The delay in getting all your responses is the price that must be paid to allow for consistent communications in the large, dense network.

Implementing the Framework in Your Application

Using this framework in the context of your own application is an easy thing. You don't need to do anything more than include the following line at the beginning of your own SNAPpy script:

```
From polling_framework import *
```


and then, make sure you call the pf_timer(ms) function from your own function hooked to the 100 ms timer (or enable the 100 ms timer hook on the pf_timer() function directly), and call the pf_setup() function from your own function hooked to the HOOK_STARTUP hook (or enable the startup hook on the pf_setup() function directly).

Your application then multicasts `ping(sequence_number, data_function)` with a TTL of 1 to perform network discovery, and `get_data(sequenced_number, target_address, function_name)` with a TTL of 1 to request data from an individual node in the network.

As an aside here, notice that the ping() command allows the application to specify a data_function as part of its signature. If a data function is specified there (as a string), that function will be called on each node as it is discovered, and the return value of that function will be included along with the node's address when it sends its own route_ping() response back toward the application. If no data function call is necessary during network discovery, pass False for the data_function parameter.

In order to interpret the responses from these calls, your PC application must also implement tell_ping() and get_data_result() functions, with the following function signatures:

`def tell_ping(sequence_number, responding_nodes, ping_data_func)`

- sequence_number is an integer representing the sequence of the ping initially sent by the PC application, so the application can tell if it is receiving a response to an "old" request, or its most recent network discovery request.
- responding_nodes is a string containing the response(s) from any node(s) found in the network. If ping_data_func is False, the responding_nodes parameter will be a concatenation of three-character SNAP network addresses of any nodes that replied as part of this ping response. If ping_data_func is True, responding_nodes will be a concatenation of one or more substrings in the following format:
  - three bytes representing the SNAP network address of the responding node
  - one byte representing the length of the data field returned from the data function specified in the PC application's original ping() message
  - a string representation of the return value of the function specified in the original ping() message
- ping_data_func is a Boolean value.

`def get_data_result(sequence_number, address, data, get_data_more)`

- sequence_number is an integer representing the sequence of this collection of data. If the data function called only returns one data packet, this will be the sequence number specified when the PC application made the initial get_data() call. Otherwise, it will be an incremented number from there for each subsequent message sent by the target node to return data.
- address is a three-character string representing the SNAP network address of the node providing the data.
- data is a string representation of the data returned by the node.
- get_data_more is a Boolean value indicating whether the PC application should expect more messages providing additional data in response to this request.

From there, when your PC application multcasts the ping() function with a TTL of 1 it will receive one or more tell_ping() messages in response, from which it can harvest its list of nodes in the network (along with additional information about the nodes if a ping_data_func was specified); and it multicasts a get-data() function with a TTL of 1 any time it wants to request information from a specific node, receiving the information in one or more get_data_result() messages.

A Sample Polling Framework Implementation

Below is the code from the sample polling framework implementation used above, with comments and discussion dispersed throughout. You can find the complete code listing without discussion in the appendix. It should also have come as a SNAPpy code file, polling_framework.py, packaged with this document.

Start by importing a few files of SNAPpy constants to facilitate referring to SNAP infrastructure.

```
from synapse.nvparams import *
from synapse.switchboard import *
```

Next, initialize all the global variables and constants used throughout the code.

```
is_bridge = False
last_seq = 0

is_get_data = False
saw_data_ack = True
data_func_name = ''
get_data_arg_count = 0
get_data_arg1 = None
get_data_addr = ''
get_data_resp = ''
get_data_more = False
sent_get_data_resp = True
get_data_wait_cntr = 0
get_data_sent_cnt = 0
GET_DATA_WAIT_CNTR_MAX = 3
MAX_GET_DATAS = 255

ping_resp_addr = '' # should be small string
responding_addrs = '' # should be a medium string
responding_addrs_index = 0
unackd_ping_ctr = 0
waiting_for_ack = False
my_rp_seq = 0
wait_cntr = 0
noise_cntr = 0
initial_delay_cntr = 0
ttl = 1
sent_addresses = True
ping_data_func = None

MAX_ROUTE_PINGS = 255
MAX_TLL = 5
```



```
ESCALATION_THRESH = 20
WAIT_CNTR_MAX = 5
MAX_PKTS = 4
INITIAL_DELAY = 3

MCAST_GROUP = 0x100

PF_VERSION = 5
```

The get_data() function is a “pretty” interface to the get_data_common() function, facilitating calls that do not include any arguments.

```
def get_data(seq, addr, func_name):
    """Calls the requested function and responds with the return data"""
    get_data_common(seq, addr, func_name, 0, None)
```

The get_data1() function is a “pretty” interface to the get_data_common() function, facilitating calls that include one argument.

```
def get_data1(seq, addr, func_name, arg1):
    """Calls the requested function with one argument and responds with the
return data"""
    get_data_common(seq, addr, func_name, 1, arg1)
```

The get_data_common() function provides a single function that can be called with either zero or one argument. If you need to be able to pass more than one parameter to a function in your network, you can modify this function (plus the two previous functions) to accommodate the number of arguments you require.

```
def get_data_common(seq, addr, func_name, arg_count, arg1):
    global is_get_data
    global last_seq
    global get_data_addr
    global data_func_name
    global saw_data_ack
    global sent_get_data_resp
    global get_data_resp
    global get_data_wait_cntr
    global get_data_sent_cnt
    global get_data_arg1
    global get_data_arg_count
    global get_data_more

    is_get_data = True
    seq &= 0x7fff
    if seq > last_seq or seq == 0:
        clear_ping() # Can't do a ping and get_data at the same time

        saw_data_ack = False
        get_data_resp = ''
        get_data_more = False
        sent_get_data_resp = False
        get_data_arg_count = arg_count
        get_data_arg1 = arg1
        if addr == localAddr():
```



```
            if arg_count == 0:
                get_data_resp = func_name()
            elif arg_count == 1:
                get_data_resp = func_name(arg1)
            saw_data_ack = True
            if is_bridge:
                sent_get_data_resp = rpc(rpcSourceAddr(), 'get_data_result',
seq, addr, get_data_resp, get_data_more)
                if sent_get_data_resp:
                    return
            else:
                sent_get_data_resp = mcastRpc(MCAST_GROUP, 1, 'get_data_ack',
seq, addr, get_data_resp, get_data_more)

        get_data_addr = addr
        if is_bridge:
            # Bridge saves requester address in get_data_addr
            get_data_addr += rpcSourceAddr()
        data_func_name = func_name
        last_seq = seq
        get_data_wait_cntr = 0
        get_data_sent_cnt = 0

        if is_bridge:
            # Go ahead and resend right away
            if arg_count == 0:
                mcastRpc(MCAST_GROUP, 1, 'get_data', seq, addr, func_name)
            elif arg_count == 1:
                mcastRpc(MCAST_GROUP, 1, 'get_data1', seq, addr, func_name,
arg1)
    else:
        sent_get_data_resp = False # resend ACK

def get_data_ack(seq, addr, resp, more):
    global saw_data_ack, get_data_addr, get_data_resp, sent_get_data_resp,
get_data_more

    if seq >= last_seq:
        if seq > last_seq or not saw_data_ack:
            get_data_more = more
            if is_bridge:
                # Bridge saves address in get_data_addr
                sent_get_data_resp = rpc(get_data_addr[3:6],
'get_data_result', seq, addr, resp, get_data_more)
            else:
                sent_get_data_resp = mcastRpc(MCAST_GROUP, 1, 'get_data_ack',
seq, addr, resp, get_data_more)
        saw_data_ack = True
        if not is_bridge:
            get_data_addr = addr
        elif get_data_more:
            check_get_data_more()
        get_data_resp = resp

def check_get_data_more():
    global last_seq, saw_data_ack
```



```
    if get_data_more and sent_get_data_resp:
        last_seq += 1
        sent = False
        if get_data_arg_count == 0:
            sent = mcastRpc(MCAST_GROUP, 1, 'get_data', last_seq,
get_data_addr[0:3], data_func_name)
        elif get_data_arg_count == 1:
            sent = mcastRpc(MCAST_GROUP, 1, 'get_data1', last_seq,
get_data_addr[0:3], data_func_name, get_data_arg1)
        if sent:
            saw_data_ack = False

def clear_ping():
    global responding_addrs, responding_addrs_index, ping_resp_addr,
ping_data_func, sent_addresses

    responding_addrs_index = 0
    ping_resp_addr = None
    responding_addrs = ''
    ping_data_func = None
    sent_addresses = True

def clear_get_data():
    global is_get_data, get_data_addr, data_func_name, get_data_arg_count,
get_data_arg1

    is_get_data = False
    get_data_addr = None
    data_func_name = None
    get_data_arg_count = 0
    get_data_arg1 = None

#@setHook(HOOK_STARTUP)
def pf_setup():
    global is_bridge
    needs_reboot = False

    if getInfo(4) == 0:
        # If this is a debug build send out error messages
        crossConnect(DS_ERROR, DS_TRANSPARENT)
        mcastSerial(1, loadNvParam(NV_MESH_MAX_HOPLIMIT_ID))

    if True:
        if loadNvParam(NV_CARRIER_SENSE_ID) != True:
            saveNvParam(NV_CARRIER_SENSE_ID, True)
            needs_reboot = True
        if loadNvParam(NV_COLLISION_DETECT_ID) != False:
            saveNvParam(NV_COLLISION_DETECT_ID, False)
            needs_reboot = True
        if loadNvParam(NV_COLLISION_AVOIDANCE_ID) != False:
            saveNvParam(NV_COLLISION_AVOIDANCE_ID, False)
            needs_reboot = True

    device_type = loadNvParam(NV_DEVICE_TYPE_ID)
    if device_type and device_type == 'Bridge':
        if loadNvParam(NV_MESH_ROUTE_AGE_MAX_TIMEOUT_ID) != 0:
            saveNvParam(NV_MESH_ROUTE_AGE_MAX_TIMEOUT_ID, 0)
```



```
            needs_reboot = True
        if loadNvParam(NV_GROUP_INTEREST_MASK_ID) != 1 + MCAST_GROUP:
            saveNvParam(NV_GROUP_INTEREST_MASK_ID, 1 + MCAST_GROUP)
            needs_reboot = True
        is_bridge = True
    else:
        if loadNvParam(NV_GROUP_INTEREST_MASK_ID) != 1 + MCAST_GROUP:
            saveNvParam(NV_GROUP_INTEREST_MASK_ID, 1 + MCAST_GROUP)
            needs_reboot = True
        if loadNvParam(NV_GROUP_FORWARDING_MASK_ID) != 1 + MCAST_GROUP:
            saveNvParam(NV_GROUP_FORWARDING_MASK_ID, 1 + MCAST_GROUP)
            needs_reboot = True

        # Turn off sending multi-casts over Packet Serial
        crossConnect(DS_PACKET_SERIAL, DS_NULL)
        is_bridge = False

    if needs_reboot:
        reboot()

    return needs_reboot

def ping(seq, data_func):
    """Global ping request"""
    global ping_resp_addr, last_seq, responding_addrs, outstanding_ping,
responding_addrs_index
    global unackd_ping_ctr, waiting_for_ack, my_rp_seq
    global wait_cntr, noise_cntr, ttl, initial_delay_cntr
    global sent_addresses, ping_data_func

    # Check if we have seen this ping before
    if seq != last_seq or seq == 0:
        clear_get_data() # Can't do a ping and get_data at the same time
        if is_bridge and seq == 0:
            if last_seq > -1:
                last_seq = -1
            else:
                last_seq -= 1
        else:
            last_seq = seq

        # Send our response to the ping back to who we originally heard it
from
        ping_resp_addr = rpcSourceAddr()

        # Setup to send our ping response when timer fires
        waiting_for_ack = False
        unackd_ping_ctr = 0
        my_rp_seq = 0
        wait_cntr = 0
        noise_cntr = 0
        ttl = 1
        getStat(9) # Reset radio counter
        sent_addresses = False
        ping_data_func = data_func
        if data_func:
            # Call function and add reponse
```



```
            responding_addrs = str(data_func())
            responding_addrs = localAddr() + chr(len(responding_addrs)) +
responding_addrs
        else:
            responding_addrs = localAddr() # Add ourselves to the list of
addresses responding to the ping
        responding_addrs_index = len(responding_addrs)

        # Rebroadcast ping request
        if is_bridge:
            mcastRpc(MCAST_GROUP, 1, 'ping', last_seq, ping_data_func)
            initial_delay_cntr = INITIAL_DELAY
        else:
            initial_delay_cntr = 0

def route_ping(seq, next_hop, addrs, rp_seq, orig_ttl, data_func):
    """Called when a node needs a ping response routed"""
    global waiting_for_ack, responding_addrs, sent_addresses

    if next_hop == localAddr() and seq == last_seq:
        # Someone needs us to forward their ping response
        if (len(responding_addrs) + len(addrs)) <= 62: # Check to make sure
there is room
            queued = mcastRpc(MCAST_GROUP, orig_ttl, 'rpA', seq,
rpcSourceAddr(), rp_seq, data_func)

            # Only add addrs if we can send an ACK, otherwise it will just
get added again
            if queued:
                responding_addrs += addrs
                if is_bridge:
                    if rpc(ping_resp_addr, 'tell_ping', last_seq,
responding_addrs, True if ping_data_func else False):
                        responding_addrs = ''
                        sent_addresses = True
    elif not is_bridge:
        ping(seq, data_func) # Check and make sure we have not seen this ping
before

def rpA(seq, addr, rp_seq, data_func):
    """Route Ping ACK"""
    global waiting_for_ack, responding_addrs, unackd_ping_ctr, my_rp_seq

    if addr == localAddr() and seq == last_seq:
        # Someone just ACK'd our route ping request
        if rp_seq == my_rp_seq:
            responding_addrs = responding_addrs[responding_addrs_index:] #
Remove addrs that we just sent but keep others
            waiting_for_ack = False
            unackd_ping_ctr = 0
            my_rp_seq += 1
    elif not is_bridge:
        ping(seq, data_func) # Check and make sure we have not seen this ping
before

#@setHook(HOOK_100MS)
def pf_timer(ms):
```



```
    global waiting_for_ack, unackd_ping_ctr, responding_addrs,
responding_addrs_index, wait_cntr
    global noise_cntr, initial_delay_cntr, ttl
    global saw_data_ack, sent_get_data_resp, get_data_wait_cntr,
get_data_sent_cnt, is_get_data

    if is_get_data:
        if get_data_sent_cnt > MAX_GET_DATAS:
            is_get_data = False
        elif not saw_data_ack:
            get_data_wait_cntr += 1
            if get_data_wait_cntr > GET_DATA_WAIT_CNTR_MAX:
                sent = False
                if get_data_arg_count == 0:
                    sent = mcastRpc(MCAST_GROUP, 1, 'get_data', last_seq,
get_data_addr[0:3], data_func_name)
                elif get_data_arg_count == 1:
                    sent = mcastRpc(MCAST_GROUP, 1, 'get_data1', last_seq,
get_data_addr[0:3], data_func_name, get_data_arg1)
                if sent:
                    get_data_wait_cntr = 0
                    get_data_sent_cnt += 1
        elif not sent_get_data_resp:
            sent_get_data_resp = mcastRpc(MCAST_GROUP, 1, 'get_data_ack',
last_seq, get_data_addr[0:3], get_data_resp, get_data_more)
        elif is_bridge and get_data_more:
            #check_get_data_more()
            saw_data_ack = False
    elif responding_addrs:
        if is_bridge:
            if rpc(ping_resp_addr, 'tell_ping', last_seq, responding_addrs,
True if ping_data_func else False):
                responding_addrs = ''
            else:
                unackd_ping_ctr += 1
        elif initial_delay_cntr > INITIAL_DELAY:
            if not waiting_for_ack:
                responding_addrs_index = len(responding_addrs)

            queued = False
            if unackd_ping_ctr > ESCALATION_THRESH and ttl < MAX_TLL:
                ttl += 1
                unackd_ping_ctr = 0
            if ttl > 1:
                wait_cntr += 1
            else:
                wait_cntr = WAIT_CNTR_MAX+1

            if wait_cntr > WAIT_CNTR_MAX:
                radio_recv_buffs = getStat(9) # auto-clears
                if radio_recv_buffs < MAX_PKTS:
                    wait_cntr = 0
                    queued = mcastRpc(MCAST_GROUP, ttl, 'route_ping',
last_seq, ping_resp_addr, responding_addrs[:responding_addrs_index],
my_rp_seq, ttl, ping_data_func)
                else:
                    noise_cntr += 1
```



```
            if queued:
                waiting_for_ack = True
                unackd_ping_ctr += 1
                if unackd_ping_ctr > MAX_ROUTE_PINGS:
                    clear_ping()
        else:
            initial_delay_cntr += 1
    elif is_bridge and not sent_addresses:
        mcastRpc(MCAST_GROUP, 1, 'ping', last_seq, ping_data_func)
```


References

Portal Reference Manual
Portal Primer

Appendix: Code Listing

```
from synapse.nvparams import *
from synapse.switchboard import *


is_bridge = False
last_seq = 0

is_get_data = False
saw_data_ack = True
data_func_name = ''
get_data_arg_count = 0
get_data_arg1 = None
get_data_addr = ''
get_data_resp = ''
get_data_more = False
sent_get_data_resp = True
get_data_wait_cntr = 0
get_data_sent_cnt = 0
GET_DATA_WAIT_CNTR_MAX = 3
MAX_GET_DATAS = 255
```



```
ping_resp_addr = '' # should be small string
responding_addrs = '' # should be a medium string
responding_addrs_index = 0
unackd_ping_ctr = 0
waiting_for_ack = False
my_rp_seq = 0
wait_cntr = 0
noise_cntr = 0
initial_delay_cntr = 0
ttl = 1
sent_addresses = True
ping_data_func = None

MAX_ROUTE_PINGS = 255
MAX_TLL = 5
ESCALATION_THRESH = 20
WAIT_CNTR_MAX = 5
MAX_PKTS = 4
INITIAL_DELAY = 3

MCAST_GROUP = 0x100

PF_VERSION = 5

def get_data(seq, addr, func_name):
    """Calls the requested function and responds with the return data"""
    get_data_common(seq, addr, func_name, 0, None)

def get_data1(seq, addr, func_name, arg1):
    """Calls the requested function with one argument and responds with the
return data"""
    get_data_common(seq, addr, func_name, 1, arg1)

def get_data_common(seq, addr, func_name, arg_count, arg1):
    global is_get_data
    global last_seq
    global get_data_addr
    global data_func_name
    global saw_data_ack
    global sent_get_data_resp
    global get_data_resp
    global get_data_wait_cntr
    global get_data_sent_cnt
    global get_data_arg1
    global get_data_arg_count
    global get_data_more

    is_get_data = True
    seq &= 0x7fff
    if seq > last_seq or seq == 0:
        clear_ping() # Can't do a ping and get_data at the same time

        saw_data_ack = False
        get_data_resp = ''
        get_data_more = False
        sent_get_data_resp = False
        get_data_arg_count = arg_count
```



```
        get_data_arg1 = arg1
        if addr == localAddr():
            if arg_count == 0:
                get_data_resp = func_name()
            elif arg_count == 1:
                get_data_resp = func_name(arg1)
            saw_data_ack = True
            if is_bridge:
                sent_get_data_resp = rpc(rpcSourceAddr(), 'get_data_result',
seq, addr, get_data_resp, get_data_more)
                if sent_get_data_resp:
                    return
            else:
                sent_get_data_resp = mcastRpc(MCAST_GROUP, 1, 'get_data_ack',
seq, addr, get_data_resp, get_data_more)

        get_data_addr = addr
        if is_bridge:
            # Bridge saves requester address in get_data_addr
            get_data_addr += rpcSourceAddr()
        data_func_name = func_name
        last_seq = seq
        get_data_wait_cntr = 0
        get_data_sent_cnt = 0

        if is_bridge:
            # Go ahead and resend right away
            if arg_count == 0:
                mcastRpc(MCAST_GROUP, 1, 'get_data', seq, addr, func_name)
            elif arg_count == 1:
                mcastRpc(MCAST_GROUP, 1, 'get_data1', seq, addr, func_name,
arg1)
    else:
        sent_get_data_resp = False # resend ACK

def get_data_ack(seq, addr, resp, more):
    global saw_data_ack, get_data_addr, get_data_resp, sent_get_data_resp,
get_data_more

    if seq >= last_seq:
        if seq > last_seq or not saw_data_ack:
            get_data_more = more
            if is_bridge:
                # Bridge saves address in get_data_addr
                sent_get_data_resp = rpc(get_data_addr[3:6],
'get_data_result', seq, addr, resp, get_data_more)
            else:
                sent_get_data_resp = mcastRpc(MCAST_GROUP, 1, 'get_data_ack',
seq, addr, resp, get_data_more)
        saw_data_ack = True
        if not is_bridge:
            get_data_addr = addr
        elif get_data_more:
            check_get_data_more()
        get_data_resp = resp

def check_get_data_more():
```



```
    global last_seq, saw_data_ack

    if get_data_more and sent_get_data_resp:
        last_seq += 1
        sent = False
        if get_data_arg_count == 0:
            sent = mcastRpc(MCAST_GROUP, 1, 'get_data', last_seq,
get_data_addr[0:3], data_func_name)
        elif get_data_arg_count == 1:
            sent = mcastRpc(MCAST_GROUP, 1, 'get_data1', last_seq,
get_data_addr[0:3], data_func_name, get_data_arg1)
        if sent:
            saw_data_ack = False

def clear_ping():
    global responding_addrs, responding_addrs_index, ping_resp_addr,
ping_data_func, sent_addresses

    responding_addrs_index = 0
    ping_resp_addr = None
    responding_addrs = ''
    ping_data_func = None
    sent_addresses = True

def clear_get_data():
    global is_get_data, get_data_addr, data_func_name, get_data_arg_count,
get_data_arg1

    is_get_data = False
    get_data_addr = None
    data_func_name = None
    get_data_arg_count = 0
    get_data_arg1 = None

#@setHook(HOOK_STARTUP)
def pf_setup():
    global is_bridge
    needs_reboot = False

    if getInfo(4) == 0:
        # If this is a debug build send out error messages
        crossConnect(DS_ERROR, DS_TRANSPARENT)
        mcastSerial(1, loadNvParam(NV_MESH_MAX_HOPLIMIT_ID))

    if True:
        if loadNvParam(NV_CARRIER_SENSE_ID) != True:
            saveNvParam(NV_CARRIER_SENSE_ID, True)
            needs_reboot = True
        if loadNvParam(NV_COLLISION_DETECT_ID) != False:
            saveNvParam(NV_COLLISION_DETECT_ID, False)
            needs_reboot = True
        if loadNvParam(NV_COLLISION_AVOIDANCE_ID) != False:
            saveNvParam(NV_COLLISION_AVOIDANCE_ID, False)
            needs_reboot = True

    device_type = loadNvParam(NV_DEVICE_TYPE_ID)
    if device_type and device_type == 'Bridge':
```



```
        if loadNvParam(NV_MESH_ROUTE_AGE_MAX_TIMEOUT_ID) != 0:
            saveNvParam(NV_MESH_ROUTE_AGE_MAX_TIMEOUT_ID, 0)
            needs_reboot = True
        if loadNvParam(NV_GROUP_INTEREST_MASK_ID) != 1 + MCAST_GROUP:
            saveNvParam(NV_GROUP_INTEREST_MASK_ID, 1 + MCAST_GROUP)
            needs_reboot = True
        is_bridge = True
    else:
        if loadNvParam(NV_GROUP_INTEREST_MASK_ID) != 1 + MCAST_GROUP:
            saveNvParam(NV_GROUP_INTEREST_MASK_ID, 1 + MCAST_GROUP)
            needs_reboot = True
        if loadNvParam(NV_GROUP_FORWARDING_MASK_ID) != 1 + MCAST_GROUP:
            saveNvParam(NV_GROUP_FORWARDING_MASK_ID, 1 + MCAST_GROUP)
            needs_reboot = True

        # Turn off sending multi-casts over Packet Serial
        crossConnect(DS_PACKET_SERIAL, DS_NULL)
        is_bridge = False

    if needs_reboot:
        reboot()

    return needs_reboot

def ping(seq, data_func):
    """Global ping request"""
    global ping_resp_addr, last_seq, responding_addrs, outstanding_ping,
responding_addrs_index
    global unackd_ping_ctr, waiting_for_ack, my_rp_seq
    global wait_cntr, noise_cntr, ttl, initial_delay_cntr
    global sent_addresses, ping_data_func

    # Check if we have seen this ping before
    if seq != last_seq or seq == 0:
        clear_get_data() # Can't do a ping and get_data at the same time
        if is_bridge and seq == 0:
            if last_seq > -1:
                last_seq = -1
            else:
                last_seq -= 1
        else:
            last_seq = seq

        # Send our response to the ping back to who we originally heard it
from
        ping_resp_addr = rpcSourceAddr()

        # Setup to send our ping response when timer fires
        waiting_for_ack = False
        unackd_ping_ctr = 0
        my_rp_seq = 0
        wait_cntr = 0
        noise_cntr = 0
        ttl = 1
        getStat(9) # Reset radio counter
        sent_addresses = False
        ping_data_func = data_func
```



```
        if data_func:
            # Call function and add reponse
            responding_addrs = str(data_func())
            responding_addrs = localAddr() + chr(len(responding_addrs)) +
responding_addrs
        else:
            responding_addrs = localAddr() # Add ourselves to the list of
addresses responding to the ping
        responding_addrs_index = len(responding_addrs)

        # Rebroadcast ping request
        if is_bridge:
            mcastRpc(MCAST_GROUP, 1, 'ping', last_seq, ping_data_func)
            initial_delay_cntr = INITIAL_DELAY
        else:
            initial_delay_cntr = 0

def route_ping(seq, next_hop, addrs, rp_seq, orig_ttl, data_func):
    """Called when a node needs a ping response routed"""
    global waiting_for_ack, responding_addrs, sent_addresses

    if next_hop == localAddr() and seq == last_seq:
        # Someone needs us to forward their ping response
        if (len(responding_addrs) + len(addrs)) <= 62: # Check to make sure
there is room
            queued = mcastRpc(MCAST_GROUP, orig_ttl, 'rpA', seq,
rpcSourceAddr(), rp_seq, data_func)

            # Only add addrs if we can send an ACK, otherwise it will just
get added again
            if queued:
                responding_addrs += addrs
                if is_bridge:
                    if rpc(ping_resp_addr, 'tell_ping', last_seq,
responding_addrs, True if ping_data_func else False):
                        responding_addrs = ''
                        sent_addresses = True
    elif not is_bridge:
        ping(seq, data_func) # Check and make sure we have not seen this ping
before

def rpA(seq, addr, rp_seq, data_func):
    """Route Ping ACK"""
    global waiting_for_ack, responding_addrs, unackd_ping_ctr, my_rp_seq

    if addr == localAddr() and seq == last_seq:
        # Someone just ACK'd our route ping request
        if rp_seq == my_rp_seq:
            responding_addrs = responding_addrs[responding_addrs_index:] #
Remove addrs that we just sent but keep others
            waiting_for_ack = False
            unackd_ping_ctr = 0
            my_rp_seq += 1
    elif not is_bridge:
        ping(seq, data_func) # Check and make sure we have not seen this ping
before
```



```
#@setHook(HOOK_100MS)
def pf_timer(ms):
    global waiting_for_ack, unackd_ping_ctr, responding_addrs,
responding_addrs_index, wait_cntr
    global noise_cntr, initial_delay_cntr, ttl
    global saw_data_ack, sent_get_data_resp, get_data_wait_cntr,
get_data_sent_cnt, is_get_data

    if is_get_data:
        if get_data_sent_cnt > MAX_GET_DATAS:
            is_get_data = False
        elif not saw_data_ack:
            get_data_wait_cntr += 1
            if get_data_wait_cntr > GET_DATA_WAIT_CNTR_MAX:
                sent = False
                if get_data_arg_count == 0:
                    sent = mcastRpc(MCAST_GROUP, 1, 'get_data', last_seq,
get_data_addr[0:3], data_func_name)
                elif get_data_arg_count == 1:
                    sent = mcastRpc(MCAST_GROUP, 1, 'get_data1', last_seq,
get_data_addr[0:3], data_func_name, get_data_arg1)
                if sent:
                    get_data_wait_cntr = 0
                    get_data_sent_cnt += 1
        elif not sent_get_data_resp:
            sent_get_data_resp = mcastRpc(MCAST_GROUP, 1, 'get_data_ack',
last_seq, get_data_addr[0:3], get_data_resp, get_data_more)
        elif is_bridge and get_data_more:
            #check_get_data_more()
            saw_data_ack = False
    elif responding_addrs:
        if is_bridge:
            if rpc(ping_resp_addr, 'tell_ping', last_seq, responding_addrs,
True if ping_data_func else False):
                responding_addrs = ''
            else:
                unackd_ping_ctr += 1
        elif initial_delay_cntr > INITIAL_DELAY:
            if not waiting_for_ack:
                responding_addrs_index = len(responding_addrs)

            queued = False
            if unackd_ping_ctr > ESCALATION_THRESH and ttl < MAX_TLL:
                ttl += 1
                unackd_ping_ctr = 0
            if ttl > 1:
                wait_cntr += 1
            else:
                wait_cntr = WAIT_CNTR_MAX+1

            if wait_cntr > WAIT_CNTR_MAX:
                radio_recv_buffs = getStat(9) # auto-clears
                if radio_recv_buffs < MAX_PKTS:
                    wait_cntr = 0
                    queued = mcastRpc(MCAST_GROUP, ttl, 'route_ping',
last_seq, ping_resp_addr, responding_addrs[:responding_addrs_index],
my_rp_seq, ttl, ping_data_func)
```



```
            else:
                noise_cntr += 1

            if queued:
                waiting_for_ack = True
                unackd_ping_ctr += 1
                if unackd_ping_ctr > MAX_ROUTE_PINGS:
                    clear_ping()
        else:
            initial_delay_cntr += 1
    elif is_bridge and not sent_addresses:
        mcastRpc(MCAST_GROUP, 1, 'ping', last_seq, ping_data_func)
```


License governing any code samples presented in this Application Note

Redistribution of code and use in source and binary forms, with or without modification, are permitted provided that it retains the copyright notice, operates only on SNAP® networks, and the paragraphs below in the documentation and/or other materials are provided with the distribution:

Neither the name of Synapse nor the names of contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SYNAPSE AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SYNAPSE OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE THIS SOFTWARE, EVEN IF SYNAPSE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Disclaimers

Information contained in this Application Note is provided in connection with Synapse products and services and is intended solely to assist its customers. Synapse reserves the right to make changes at any time and without notice. Synapse assumes no liability whatsoever for the contents of this Application Note or the redistribution as permitted by the foregoing Limited License. The terms and conditions governing the sale or use of Synapse products is expressly contained in the Synapse's Terms and Condition for the sale of those respective products.

Synapse retains the right to make changes to any product specification at any time without notice or liability to prior users, contributors, or recipients of redistributed versions of this Application Note. Errata should be checked on any product referenced.

Synapse and the Synapse logo are registered trademarks of Synapse. All other trademarks are the property of their owners.

For further information on any Synapse product or service, contact us at:

Synapse Wireless, Inc.
500 Discovery Drive
Huntsville, Alabama 35806

256-852-7888
877-982-7888
256-852-7862 (fax)

www.synapse-wireless.com

