A Lightweight Networking Strategy for Robotic Control Systems

Jake Read
Massachusetts Institute of Technology
Cambridge, Massachusetts
jakeread@mit.edu

Douglas Kogut
Massachusetts Institute of Technology
Cambridge, Massachusetts
dkogut@mit.edu

Nicholas Selby
Massachusetts Institute of Technology
Cambridge, Massachusetts
nselby@mit.edu

Patrick Wahl
Massachusetts Institute of Technology
Cambridge, Massachusetts
pwahl@mit.edu

ABSTRACT

We present an open-source networking protocol for use in Networked Control Systems that can be implemented on system endpoints trivially and runs with no global state, allowing for enhanced memory conservation and fault tolerance. This strategy, which we call *TinyNet*, allows for real-time multipath routing by maintaining a table at each router that tracks which of the router's ports is part of the shortest path to each known destination node. TinyNet uses a busyness-metric based cost function, taking the buffer size at each neighbor node to avoid routing bottlenecks and ensure optimal network utilization. TinyNet's small packet sizes allow for efficient transmission of data, making it ideal for simple network models that primarily involve maintaining control loops. [insert stuff about performance data here]

CCS CONCEPTS

• **Networks** → **Network protocol design**; **Routing protocols**;

ACM Reference format:
Jake Read, Douglas Kogut, Nicholas Selby, and Patrick Wahl. 2017. A Lightweight Networking Strategy for Robotic Control Systems. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 8 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Networked control systems (NCS) are critical parts of complex robotics and avionics systems where many sensors and actuators work together to perform a common goal (e.g. locomotion, stabilization, sensor fusion, position control, etc.). NCS are also often employed in manufacturing, where multiple machines are linked to coordinate material handling and production scheduling. The field of NCS is unique from other networking fields in important ways.

- **Total throughput is valued but is not a key metric.** Rather, message sizes are typically very small (between three and fifty bytes) and message delay time is the critical metric. Often, messages are only one-packet in length.
- **Determinism in message delivery time is critical.** Systems must guarantee that certain control loops "close" within a defined time lest they become unstable. This means that the distribution of round trip times (RTT's) on the network should have a very small standard deviation.
- **Robustness is critical.** NCS should not contain any single points of failure. Additionally, NCS should not require network controllers to coordinate port-forwarding tables; the time it takes for a network controller to modify routing tables on all nodes in the network is nontrivial, and leads to large increases in the standard deviation of RTT's.

1.1 The State of the Art

The field of NCS is only recently moving away from the use of FieldBusses. State-of-the-art NCS employ simple Switched Ethernet or proprietary protocols based on Switched Ethernet in order to route traffic[4][6].

Switched Ethernet has been adopted in the NCS industry due to the accessibility of hardware and the open standard it offers - allowing control products from multiple vendors to be potentially integrated by a customer and allowing systems to increase in scale and complexity without the re-configuration or specification of a FieldBus. However, Switched Ethernet is not ideal for the particular demands of NCS and it is not likely that it will meet the demands of NCS in the near- or long-term.

1.2 The Limits of Available Technology

As we will discuss in Section 2, Switched Ethernet possesses the critical limit that a network graph may not contain more than one route to any given endpoint. This results in message bottlenecks at particular switches and network graphs that contain Single Points of Failure (SPoF) wherein a singular link failure can result in many endpoints becoming unreachable, often permanently so.

In order to increase Message Delivery Determinism, a key metric for NCS performance, we also look to provide a strategy for multipath routing. With multipath, NCS can operate with highly connected graphs rather than spanning-trees and route messages around busy nodes rather than queuing them in a long buffer.

However, we find that existing multipath routing techniques, largely developed for use in datacenters, require the use of network controllers (NC's) to update port-forwarding tables. These NC's necessarily contain information about the state of the entire network graph. During a link failure or rapid change in network

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

utilization (say, a single router suddenly becoming very busy), these network controllers are responsible for re-converging on an acceptable routing topology and forwarding new routing rules to each node in the network. This convergence happens on the order of one second [5], with a lower bound of approximately 200ms [2][3]. Because these re-convergences cause the rest of the network to shut down momentarily, they are likely to cause control loops in an NCS to fail. For this reason, existing multipath routing techniques are not appropriate for use in NCS.

1.3 TinyNet: Stateless Multipath

TinyNet is an experiment in networking strategy that combines the relative simplicity of Switched Ethernet with the advantages offered by multipath routing techniques seen in datacenters. In order to provide multipath capability without a network controller to coordinate forwarding tables, we implement a port-forwarding protocol based on each routers' *local knowledge* of its neighbours' packet queue as well as historical data on packets previously seen passing through the routers' ports. In this paper, we will outline the protocol rules we have developed for TinyNet and evaluate the protocol using embedded hardware as well as a simulated network model.

2 RELATED WORK

Related work can be divided into three distinct areas: convergence in networked systems, routing protocols, and Switched Ethernet.

2.1 Convergence in networked systems

During both startup and topology change, nodes need to establish knowledge of the network graph to a varying degree of completeness. In most routing protocols, this requires either nodes to broadcast their routing tables or nodes to send flood messages to crawl the topology manually.

One of the key features of the TinyNet protocol is the ability to react in real-time to failure scenarios, i.e. dynamically reroute paths in the event of a link/node failure. In this section, we analyze the relative ability of our protocol to reestablish a network quickly when failure scenarios happen.

Convergence refers to the reestablishment of consistency in topological information that routers have about their network. In our protocol, convergence will refer to the delay that occurs when a node is removed from the network and the subsequent dynamic packet flooding and re-routing that occurs. Note that convergence is slightly different in our protocol since each node will not have a complete view of the network graph. However, convergence time from other routing protocols will serve as good benchmarks as convergence time measures the time it takes the network to operate normally in the event of a node failure or of node busyness.

2.1.1 *Comparison to other protocols.*

We analyze the failure scenarios in 3 protocols, namely OSPF, SPB, and TRILL. In all 3 of these protocols, each node must know the entire network graph to calculate the shortest path between a source and destination. Thus, when a node goes down, the entire network will halt and update their view of the network graph, drastically decreasing message delivery time determinism. These protocols seem to imply a standard convergence time of around 200ms [2][3]. In TinyNet, when a node is withdrawn, information regarding failed links and nodes propagates naturally and efficiently through the network without altering network state or significantly increasing delay time

Heavily associated with convergence time is the ability for routing protocols to consider multiple paths. Each of the 3 protocols above offers such strategies. TRILL uses Fabric Shortest Path First (FSPF) to find an alternate route upon topology change. OSPF and SPB can be configured to use equal-cost multipath (ECMP) routing to maintain multiple paths when there exists more than one path with the same cost. These 3 protocols offer dynamic path finding only in certain scenarios. TinyNet will naturally use a greedily found path with significantly less overhead than stateful protocols.

TinyNet improves upon these protocols by decreasing packet header size and performing less computationally expensive tasks, i.e. not running Dijkstra's algorithm on a network graph. This improves the usability of the TinyNet protocol across a large range of hardware.

2.2 Switched Ethernet

In Switched Ethernet, because a minimum spanning tree is created, nodes in a particular layer compete for link-time on the layer above. Message delay time increases linearly with the probability that peers are transmitting at the same time and with the number of peers on that layer.

In addition, Switched Ethernet contains single points of failure, where a broken link or switch means that the network must re-run the spanning tree protocol algorithm - a process that often takes several seconds [5]. Because Switched Ethernet graphs are hierarchical, it is often the case that failure on a single link can cause entire sections of the network to become unreachable.

Furthermore, Switched Ethernet is non-programmable, i.e., switches are black-box integrated circuits and do not allow systems designers to arbitrarily add functions to a system on the networking layer. For example, while it is desirable in NCS to implement message priorities and load balancing, this is not possible with a Layer 2 routing strategy and must be added in the application layer.

3 THEORY

TinyNet's protocol almost entirely removes inter-node administrative communication. Instead of communicating topological data to each other, nodes only send heartbeats to their immediate neighbors and forward packets to designated ports. In this sense the protocol is stateless. During normal execution and failure scenarios, TinyNet is able to function in a distributed manner, delegating the work of sending the message to a neighbor instead of pre-planning the entire route. This also improves link utilization and goodput relative to other protocols.

We design TinyNet with the following constraints:

- **Integration in Endpoints.** We develop TinyNet such that it can be easily integrated into robotic endpoints. We use a UART link, a peripheral available on nearly any microcontroller, and write the network protocol into a C library that can be trivially attached to existing firmwares.
- **Open Source.** We develop TinyNet as an open-source project such that others developing Network Control Systems can

implement it (as above) into their projects on whichever hardware they choose. We hope that this, rather than attempting to establish a closed standard, will allow the project to take on many instantiations.

- **Statelessness.** We develop TinyNet with the goal of minimizing the centralized awareness of state needed to maintain routing. This increases overall robustness and eliminates the need for a central network controller or other network configurations.
- **Real-time Multipath.** We are motivated to deliver real-time multipathing for port-forwarding in TinyNet.
- **Robustness.** TinyNet must continue to perform in the face of link or router losses.

3.1 The TinyNet Algorithm

Each node handles incoming packets as follows:

```
if the packet is not a buffer update:
 update the LUT using packet src. and hop count

if packet is standard:
    if I am destination:
        process data in packet
        reply with ACK
    else:
        increment hop count
        if LUT has destination address:
            send packet to port which minimizes C(hops, buffer) over all ports
        else:
            send packet to all ports as standard flood

elseif packet is ACK:
    if I am destination:
        process acknowledgement
    else:
        increment hop count
        if LUT has destination address:
            send packet to port which minimizes C(hops, buffer) over all ports
        else:
            send packet to all ports as ack flood

elseif packet is standard flood:
    remove packet dest. from LUT at that port if it exists
    if I have not yet seen this flood:
        if I am destination:
            process data in packet
            reply with ACK
        else:
            increment hop count
            if LUT has destination address:
                send packet to port which minimizes C(hops, buffer)as standard
      packet
            else:
                send packet to all ports except one from which it was received

elseif packet is ACK flood
    remove packet dest. from LUT at that port if it exists
    if I am destination:
        process acknowledgement
    else:
        increment hop count
        if LUT has destination address:
            send packet to port which minimizes C(hops, buffer) as standard ACK
        else:
        send packet to all ports except one from which it was received
else:
    write buffer depth to LUT
```

In this algorithm, LUT is look-up table and ACK is acknowledgement. At a high level, the packet handling algorithm identifies whether the incoming packet is a standard message or an ACK, whether or not it is a flood, and whether or not it is a heartbeat, all by reading the first byte of the packet. Packets are structured as follows:

Type	8 b	10 b	6 b	10 b	6 b	N B
STD	0xFF or FE	Dest.	HC	Src.	B	MSG
ACK	0xFD or FC	Dest.	HC	Src.	-	-
HB	-	-	-	-	-	-

STD is standard packet, ACK is an acknowledgement, and HB is a heartbeat containing only a buffer depth between 0 and 251 so as to avoid ambiguity with the four other packet types. 0xFE or 0xFC designate the packet as a flood standard or flood acknowledgement, respectively. 0xFF or 0xFD designate the opposite. In the top row, lowercase 'b' is "bits" and uppercase 'B' is "bytes." The " B" is the length of the message payload in bytes. Dest., HC, Src., and MSG are the destination ID, hop count, source ID, and message payload, respectively.

3.2 Message Delay Determinism

One major axis along which TinyNet must perform is Message Delay Determinism, i.e. how quickly and reliably a packet can reach its destination. For a given packet at a given node, the total amount of time spent on the packet can be described as:

$$D = D_{rx} + D_{process} + D_{tx} \tag{1}$$

where D_{rx} and D_{tx} is the amount of time for the packet to be received and transmitted, respectively, over UART and $D_{process}$ is the amount of time required for processing the packet. $D_{process}$ generally remains constant for all packets which are not heartbeats and can be measured directly using a logic analyzer.

The amount of time spent receiving the packet, D_{rx}, can be computed from:

$$D_{rx} = P_l \times D_{byte} + L_{tt} \tag{2}$$

where P_1 is the number of bytes in the packet, D_{byte} is the amount of time the processor takes to process the byte in the UART buffer, and L_{tt} is the link transmission time:

$$L_{tt} = P_1 \times 10/L_{br} \tag{3}$$

where L_{br} is the bitrate over the link. Note that a byte sent over UART contains eight message bits plus one start and one stop bit for a total of ten bits. These parameters can be measured directly using a logic analyzer. Generally, D_{rx} is dominated by the bit rate term.

The amount of time spent transmitting the packet, D_{tx} is computed similarly:

$$D_{tx} = P_1 \times (N \times D_{byte} + 10/L_{br}) \tag{4}$$

where N is the number of ports along which to send the packet. Note that N does not multiply the bit rate term because transmission along each port happens simultaneously.

If multiple packets are received at a node before each one can be processed, incoming packets are stored in a queue until the processor is ready to handle them. As the frequency of incoming packets increases, so does the depth of the queue and thus the latency between packet reception and transmission. For this reason, it is important for nodes to be able to apply backpressure on the network to signal to other nodes that alternative message paths may be preferable.

TinyNet uses a periodic heartbeat to apply backpressure. At a predefined period, each node will transmit a single byte containing its current buffer depth to each neighboring node. This packet is

not forwarded and serves only to notify neighbors so each node can have a more robust path cost function of both hop count as well as a "busyness metric":

$$C(HC, BD) = HC + \lambda BD \quad (5)$$

where HC is the estimated hop count to the destination along this port and BD is the buffer depth of the node connected to this port. λ is a tunable parameter that can be set by the network designer. The hop count can be estimated from previous packets received on this port originating from the destination node. The buffer depth is determined from the heartbeat.

3.3 Robustness to Failure

Traditionally, stateful and stateless network protocols handle node failure very differently. The more the network protocol requires nodes to be aware of current network topology, the longer it takes to propagate information about node failure. Consider a worst-case scenario of a network running OSPF which requires all nodes to have full, real-time knowledge of network topology in order to perform routing. After node failure was discovered by its neighbors, a packet containing information about that disconnection would have to be propagated and processed by every other node in the network. Thus, it is desirable to reduce or remove network statefulness to improve recovery time from node failure.

However, stateless network protocols exhibit a different problem. These techniques typically initialize a spanning tree over the network, effectively cutting hardware links that create loops. Unfortunately, if a node that is part of the spanning tree fails, the branches below that node become unreachable until the network senses the node failure and regenerates the entire spanning tree.

TinyNet aims to take advantage of the multipath routing of stateful networks without incurring large recovery time. To accomplish this, TinyNet again leverages the heartbeat, this time to create a distributed understanding of network state rather than an absolute centralized one. At a period arbitrarily larger that that of expected heartbeat reception, each node pauses to ensure it has recently received a heartbeat along each port. If it has, it can assume that the node connected along that port is working. If is has not, it can assume that node has failed and update its look-up table accordingly. By "taking the pulse" of its neighbors, each node can evaluate local network state.

However, the heartbeat only notifies nodes directly connected to the failed node; TinyNet still needs to notify all nodes that would use this path to communicate. To accomplish this, each non-heartbeat packet contains a single bit identifying whether or not the packet was flooded from the transmitting node. Because nodes will only flood messages for which their look-up table does not contain a path, a node that receives a flooded message on a given port can assume that said port is not a path to the destination.

In the event of node failure, neighboring nodes will notice a sudden absence of heartbeats and remove all corresponding entries from its look-up table. Assuming that port was the only path to the destination for that node, if that node receives another packet with the same destination, it responds by flooding the packet back with the flood designation. This effectively notifies the previous node that the path is no longer available. This process repeats itself until a node is reached that knows a different path to the destination, at which point the packet ceases to be a flood and propagates along the new path as a standard packet. Thus, only the nodes that benefit from knowledge of node failure are notified, and they are notified when they need to transmit.

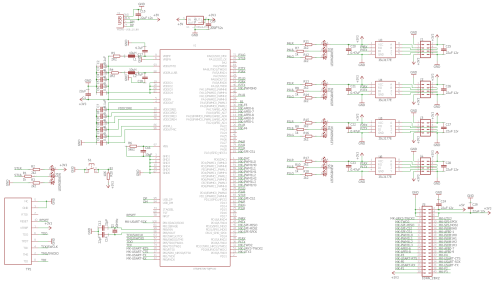

Figure 1: The router schematic.

4 IMPLEMENTATION

4.1 Hardware

We designed and fabricated small hardware routers for TinyNet. We use an ATSAMS70 microcontroller from Atmel/Microchip, running with a core clock of 300MHz. The chip is capable of transmitting UART at up to 18.75MPBS, but we found in practice we had to lower the bitrate to 3.125 MBPS before seeing reliable transmission. We use an ISL3177 differential driver on the UART lines to bring some EMI resilience to the system. Although this was likely not necessary in our testbed, our future work includes bringing the system into harsh EMI environments (i.e. nearby Brushless Motor Controllers).

4.2 Simulation

We simulate TinyNet in JavaScript on a Windows 10 Laptop PC. The code is modified from the Simbit P2P network simulator [1]. Simbit is written specifically for blockchain technologies like Bitcoin. We modified it so that the network topology is defined as an array of arrays representing the connections each node has on its various ports. This topology can be created algorithmically to enable scaling of the network to tens of thousands of nodes.

The simulation models the software running on each processor in the network as a "manager" which handles incoming requests on each of its ports. Each manager keeps track of its own look-up and buffer depth tables, and will process incoming requests using the forwarding algorithm. The managers send their actions to an asynchronously-running network controller, which facilitates communication between them. Manual simulation actions can also be fed to the managers in order to test different types of communication on the network.

5 EVALUATION

5.1 Evaluation Methods

In order to evaluate the success of TinyNet, we use our hardware testbed in order to time key message passing times and delays and

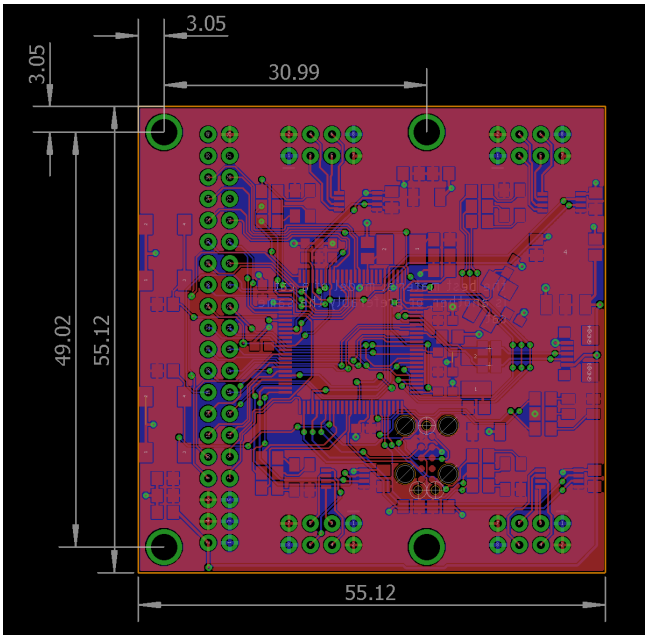

Figure 2: The router board file.

use those characterizations to inform a simulation. The simulation allows us to quickly iterate through network traffic scenarios, and measure in detail how messages are routed through the network.

5.2 Hardware

In order to evaluate our hardware systems, we used a *Saleae Logic 8* Logic Analyzer to read logic level voltages from the microcontrollers. In the application and networking layers of the router firmware, pins are turned high or low to indicate the occurrence of the events we are interested in timing.

5.2.1 *Processing Delay.*

We denote Processing Delay $D_{process}$ as the time it takes for one router to, upon receiving a packet, calculate which port to forward it along. In order to characterize this delay, we measure signals on the link layer using a logic analyzer as a packet traverses a single router. We take the time between the end of an incoming packet and the beginning of the transmission of the outgoing packet as the $D_{process}$.

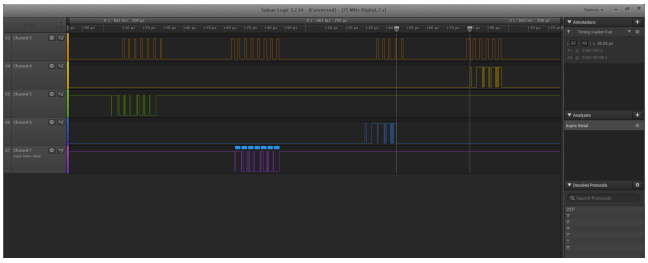

Figure 3: Measuring Packet Delay Time.

We averaged measurements from 10 packet traversals, and found an average of $37\mu s$.

5.2.2 *Link Transmission Time.*

We denote Link Transmission Time L_{tt} as the time it takes for a message to traverse a link. This is a simple calculation where $L_{tt} = P_l * 10/L_{br}$, where P_l is the length of the packet in bytes, and L_{br} is the Link Bitrate. We use a Logic Analyzer to confirm that this is the case. Additionally, transmission and reception of bytes also occupies the processor for some time. While the UART link handles bit-shifting asychronously on each port, every time a character is received the port fires an interrupt that must be addressed by the processor. We measured this interrupt handling time to be $D_{byte} = 1.5\mu s$ per Byte. Indicators for the handling of these interrupts can be seen in Fig. 3 on Channel 3. Methods for accounting for these delays such that results from our Simulation match those of our Hardware is discussed in the Implementation Section.

5.2.3 *12 Router Behavior.*

In order to characterize the performance of multi-link routing, we construct a highly connected grid of 12 TinyNet Routers. In Fig. 4 we see this grid, along with the Logic Analyzer used to time RTT, D_{packet} and verify L_{tt}. Also pictured is a TinyNet Bridge that allows the network to be accessed via a USB Port.

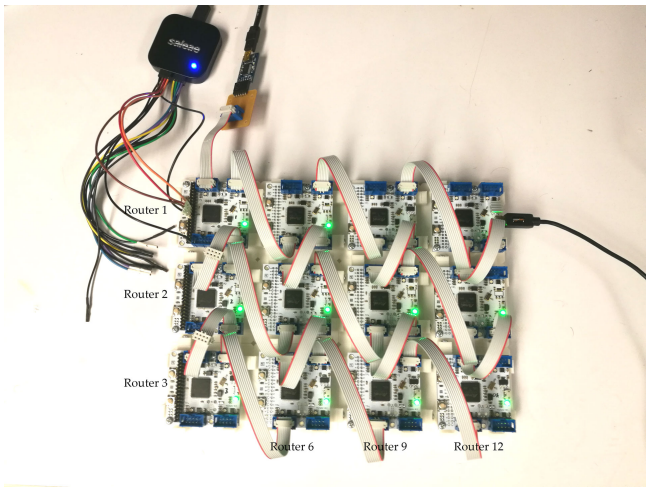

Figure 4: A Testbed of 12 TinyNet Routers.

We then measured RTT between Router 1 and Router 12. We see that one RTT is $511\mu s$.

5.3 Simulation Validation

We are able to simulate varying network topologies and traffic scenarios using our visual-equipped simulator, which runs in JavaScript and is based on the open-source Simbit architecture[1]. To evaluate the simulation, we simulate a 12-node grid structure identical to the hardware implementation and use the hardware parameters measured by the logic analyzer to initialize the nodes. After verifying the simulation accurately models the real world hardware, we can use the simulation to run large-scale experiments. These

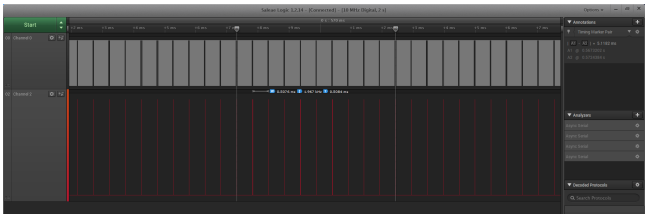

Figure 5: A 511μs RTT with 12 routers.

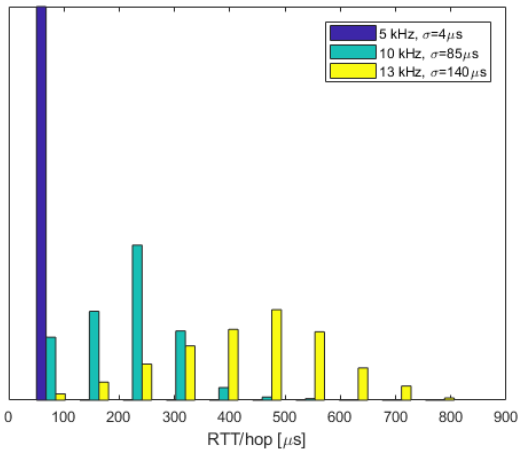

Figure 6: Histograms of Message Delay Times in a Grid with Varying Levels of Cross-Traffic.

experiments include evaluating TinyNet in the control system for a wing in avionics and other network structures.

5.3.1 12 Router Behaviour.
To verify the simulation's ability to model reality before using it to estimate performance, we construct a generic, highly-connected grid of 12 nodes. As in the hardware implementation, we transmit messages across the grid diagonal and measure the round trip time. Recalling the RTT measured across the hardware of 511 μs, the simulation reported a RTT of 505 μs corresponding to an error of 1.2 percent. Thus, we conclude that the simulation accurately models the real world system and can be used to perform larger experiments.

5.4 Determinism of Communication Time

The first axis along which TinyNet must perform is communication time determinism, the standard deviation of per hop round trip time to send a message.

To evaluate this key metric, we begin by simulating a 4-by-4 grid[1] of nodes. Each node is assumed to have a packet delay time of 30 μs and each link a bit rate of 20 MHz. We transmit important messages across one diagonal at a frequency of 5 kHz for 30 ms. Across the other diagonal, we simulate various levels of cross-traffic.

[1]Unlike past systems that construct spanning tree abstractions from their physical network topologies, TinyNet's performance is enhanced by larger network topologies. Thus, we evaluate determinism and robustness using a small grid size.

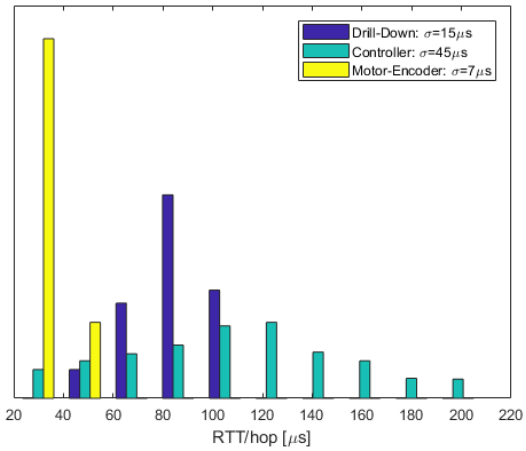

Figure 7: Histograms of Message Delay Times in an Airplane Wing Model Network.

The first simulation is run with cross-traffic messages being sent at 5 kHz, the second at 10 kHz, and the third at 13 kHz. We illustrate the histograms of the per packet round trip times for messages in Fig. 6. The standard deviations for each cross-traffic scenario are 4 μs, 85 μs, and 140 μs, respectively.

In addition to testing determinism in a grid network, we also evaluate in a simulated airplane wing model network. This network consists of four layers. The top layer is a single master node. The master node is connected to each of three controllers making up the second layer. Each controller is connected to each of eight motors making up the third layer. Each motor is connected to two neighboring motors and, in the fourth layer, a corresponding encoder. With the same hardware parameters used in the grid, communication requirements are as follows: the motors maintain a 2.5 kHz control loop with their encoders and the controllers a 1 kHz loop with every motor. Furthermore, the master node must drill down and query each encoder at 500 Hz. The simulation was run for 30 ms. The histograms of per hop round trip time are illustrated in Fig. 7. The master-encoder query has a standard deviation of 15 μs, the controller-motor loops 45 μs, and the motor-encoder loops 7 μs.

These results highlight two key outcomes:

- Increased traffic across a network reduces network determinism. As more packets travel across the network, the buffer depth of individual nodes in the network grows, increasing both message latency and the variance of that latency.
- TinyNet is able to maintain high message determinism even in networks with heavy cross-traffic. This is to be expected since TinyNet dynamically adapts message paths to increased traffic through different parts of the network. This behavior balances the load across many nodes, reducing message delivery time and improving TinyNet's ability to maintain consistently low latency.

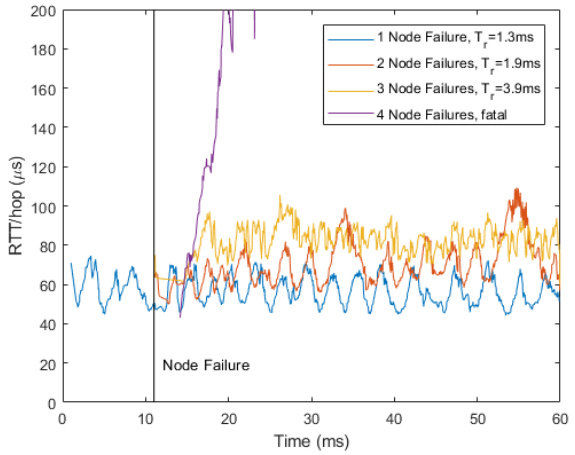

Figure 8: Message Delay Time of Grid Network Before and After Random Node Failure

5.5 Robustness to Node Failure

The second axis along which TinyNet must perform is robustness to node failure, how quickly the network can react and recover from node failure. To evaluate this key metric, we begin by simulating a 4-by-4 grid of nodes with the same hardware parameters used to evaluate message determinism. Corner nodes send messages to their partners across the diagonals with a frequency of 10 kHz and 5 kHz, respectively. After 11 ms, one, two, three, or four nodes are randomly disconnected from the network and the message delay times are recorded. The results are illustrated in Fig. 8. After a single node failure, the network corrected after 1.3 ms. After two, 1.9 ms. After three, 3.9 ms. This result highlights key findings:

- TinyNet recovers almost instantly from node failures, even when 18 percent of the network simultaneously fails. Compared to stateful techniques which can take several seconds [5] to react to lost nodes, TinyNet's millisecond-scale delay translates to dramatically fewer lost packets and virtually no effect on message round trip time.
- While the loss of a single node had no discernible impact on long-term communication time, the loss of multiple nodes negatively impacted both message latency and determinism. This is to be expected because TinyNet's load balancing can only be effective if there are more nodes along which to transmit messages.
- TinyNet is capable of maintaining message transmissions across a network even when 18 percent of the network has failed. In contrast to previous spanning tree protocols for which a single node failure can mean the loss of an entire branch of nodes, TinyNet dynamically notices and routes around lost nodes without changing state. However, there is a threshold of node failures after which network latency becomes unstable. For this set of parameters, the network failed after it lost 25 percent of its nodes.

6 CONCLUSION

This paper presents a novel routing protocol, TinyNet, developed specifically for Networked Control Systems, where Message Delivery Time Determinism, rather than Total Throughput, is to be maximized. We have demonstrated TinyNet's effectiveness in routing messages with very simple hardware and developed a simulation as a tool for evaluating the protocol's effectiveness. We have seen that TinyNet is able to route messages across highly connected graphs without the use of any Network Controller and without any node having local knowledge of the entire network graph.

Our greatest contribution is the development of a port-forwarding strategy that incorporates real-time information about the next hops' current queue size in order to intelligently re-route packets around busy areas in a network graph.

6.1 Future Work and Concluding Thoughts

We are interested in continued work on TinyNet as we believe there is a real problem solution fit at hand.

6.1.1 *Contacting Industry Experts.*

It has been difficult to ascertain what realistic Network Control Systems utilization profiles are, as there is very sparse literature on the subject. Our approach to systems development has then been limited to developing generic situations (like our interconnected grid). We have reached out to professionals at Boeing, Airbus, Kittyhawk and Moog Inc in order to begin collecting primary research data on real world uses and challenges faced in the implementation of Networked Control Systems.

6.1.2 *FPGAs for a Stateless Link Layer.*

We began work on an FPGA-based link layer that uses a technique we call 'co-clocking' to establish bitrate. With co-clocking, neither side of the transmission specifies a bitrate, rather, when each side has shifted data into a register, it 'replies' with a acknowledgement bit, triggering the next bit to be transmitted. We have seen this link demonstrate bitrates up to 65 Mbps, and have shown the FPGA communicating with a microcontroller similar to the ATSAMS70 used in the TinyNet router at similar speeds using a wide parallel bus running at 5 MHz. We show this system, in its infancy, in Fig. 9. We see that the use of UART is a weak link in the project for the reasons that (1) it limits permissible bitrate to a mere 3.125 Mbps and (2) it is 'stateful' in the sense that all ports must be configured to operate at the same sampling period, or baudrate. We would like to demonstrate a TinyNet implementation using this stateless link layer in order to demonstrate a truly stateless, configuration-free system.

6.1.3 *FPGA's for Switching.*

We are also interested in working towards FPGA based port-forwarding. We believe the TinyNet protocol is simple enough to be rendered in Verilog with only minimal expertise. We hope that this might drive our system performance objectively past Switched Ethernet, while maintaining the stateless, fault-tolerant, and adaptive multipath strategy that we have developed in this instantiation of the project.

6.1.4 *Machine Learning for a Lambda Function.*

Currently, our routing protocol implements a Lambda Function to route around busy ports as shown in Eq. 5. We would like to

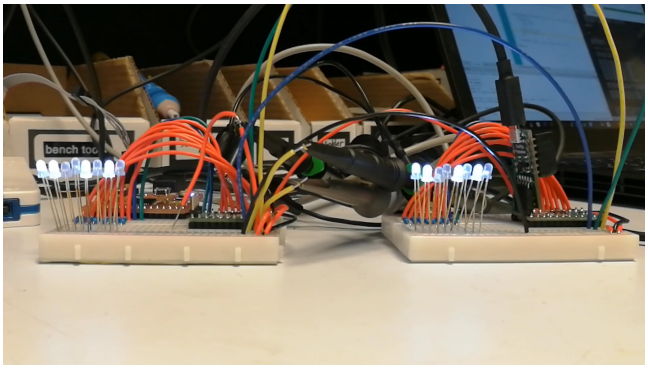

Figure 9: Our FPGA CoClocking Link Layer: four Serial lines are used to transmit data at 65MBPS between two FPGAs, which use an 8-bit wide parallel port to shift entire bytes into a microcontroller on each clock cycle.

explore using various machine learning techniques to approximate the function of RTT given network topology and cost function and use the model to optimize the cost function to improve network performance.

REFERENCES

[1] ebfull. 2016. Simbit. https://github.com/ebfull/simbit. (2016).

[2] V Eramo, M Listanti, and A Cianfrani. 2008. Multi-path OSPF performance of a software router in a link failure scenario. *Telecommunication Networking Workshop on QoS in Multiservice IP Networks, 2008. IT-NEWS 2008. 4th International* (2008). https://doi.org/10.1109/ITNEWS.2008.4488153

[3] J Farkas and Z Arato. 2009. Performance Analysis of Shortest Path Bridging Control Protocols. *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE* (2009). https://doi.org/10.1109/GLOCOM.2009.5425776

[4] Rachana Ashok Gupta and Mo-Yuen Chow. 2010. Networked Control System: Overview and Research Trends. *IEEE Transactions on Industrial Electronics* 57, 7 (July 2010), 2527–2535. http://foresight.ifmo.ru/ict/shared/files/201309/1_7.pdf

[5] Y Krishnan and G Shobhai. 2013. Performance analysis of OSPF and EIGRP routing protocols for greener internetworking. *2013 International Conference on Green High Performance Computing (ICGHPC)* (2013). https://doi.org/10.1109/ICGHPC.2013.6533929

[6] James R. Moyne and Dawn M. Tilbury. 2007. The Emergence of Industrial Control Networks for Manufacturing Control, Diagnostics, and Safety Data. *Proc. IEEE* 95, 1 (Jan. 2007), 29–47. http://www.dei.unipd.it/~schenato/didattica/PSC07/NCS_Tilbury.pdf