E PluriBus Unum: High Performance Connectivity On Buses

Ratul Mahajan Jitendra Padhye Sharad Agarwal Brian Zill
Microsoft Research

Abstract – We present PluriBus, a system to provide high-performance Internet access on-board moving vehicles. It seamlessly combines multiple wide-area wireless paths that individually tend to be lossy and high-delay. PluriBus employs *opportunistic erasure coding*, a novel technique to use spare capacity along any path to mask losses from end hosts. It sends erasure coded packets only when there is an opening in a path's spare capacity, so that coded packets do not delay or steal capacity from ordinary data packets. Packets are coded using *Evolution* codes that we have developed to greedily maximize the expected number of data packets recovered with each coded packet. We have deployed PluriBus on two buses. Our experiments show that it reduces the median flow completion time by a factor of 2.5, compared to an existing method for spreading traffic across multiple paths.

1. INTRODUCTION

Internet access on-board buses, trains, and ferries is increasingly common [33, 38, 40, 41]. Public transportation agencies in over twenty cities in the USA currently provide such access to boost ridership; many more are planning for it [32]. Corporations also provide such access on the commute vehicles for their employees [37, 39]. For instance, more than one-quarter of Google's work force in the Bay Area uses such connected buses [37]. By all accounts, riders greatly value this connectivity. It enables them to browse the Web, exchange email, and work on the way to their destinations.

Despite increasing popularity and a unique operating environment, the research community has paid insufficient attention to how to best engineer these networks. This is the focus of our work. It is motivated by our own experiences of poor performance of these networks and complaints by other users [34, 35, 36]. Based on early experiences with its commuter service, Microsoft IT warns that "there can be lapses in the backhaul coverage or system congestion" and suggests "cancel a failed download and re-try in approximately 5 minutes."

Figure 1(a) shows the typical way to enable Internet access on buses today. Riders use WiFi to connect to a device on the bus (e.g., [12]), which we call *VanProxy*. The device provides Internet access using a wide-area wireless (WWAN) technology such as EVDO or HSDPA. The key to good performance in this setup is the quality of connectivity provided by the wireless link.

Our measurements of multiple technologies confirm earlier findings [27, 13] that WWAN paths offer poor service from moving vehicles. They have high delays and frequently drop packets. Occasionally, they even suffer blackout periods with very high loss rate. Poor application performance in this setting is only to be expected.

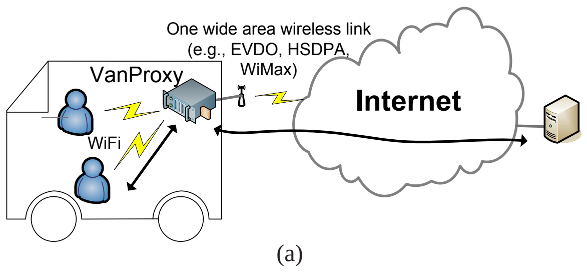
We design and deploy a system called PluriBus, to provide high-performance connectivity on-board moving vehicles. As shown in Figure 1(b), it uses multiple WWAN links and bonds them with the help of a proxy on the wired network, which we call *LanProxy*. Our approach is inspired by MAR [27] which showed the potential of using multiple wireless links using simple bonding mechanisms and trace-driven studies. It left open the task of building high-performance mechanisms.

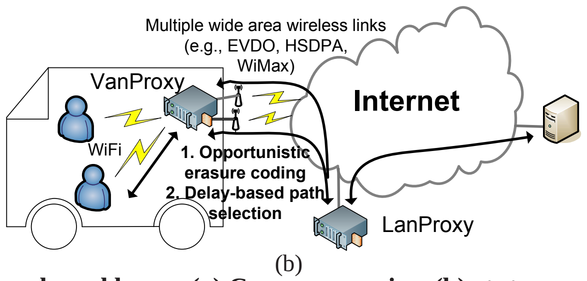
PluriBus employs two techniques that boost application performance. These techniques reduce the path loss rate and path delay experienced by end hosts. Reducing these quantities directly improves the completion times of short TCP transfers that dominate in our environment.

To mask losses from end hosts, PluriBus uses a novel technique called *opportunistic erasure coding*. It is opportunistic in when and how many erasure coded packets are sent as well as what each coded packet carries. It sends coded packets only during instantaneous openings in a bottleneck link's available capacity, which we judge by estimating queue length. This way, coded packets do not delay or steal bandwidth from data packets and provide as much protection as available capacity allows. By contrast, in existing methods the fraction of coded packets per data packet is independent of load and available capacity [2, 21]. These methods may either fail to take full advantage of available additional capacity or may slow down data packets under high load.

PluriBus encodes packets using a new rateless code called *Evolution* codes. These codes maximize the expected number of data packets recovered with each coded packet, by explicitly considering what information might already be available at the receiver. They thus aim for greedy, partial recovery of a window of packets. In contrast, traditional erasure codes, such as Reed-Solomon and LT [26, 22], aim for efficient, full recovery. They minimize the number of packets needed at the receiver to recover all data. But in our setting, given the burstiness of incoming traffic and losses, it is hard to guarantee at short time scales that the required number of packets will be received. And when that does not happen, very little data may be recovered by traditional erasure codes [28].

The second technique, called *delay-based striping*, minimizes the average packet delivery delay. We achieve this




Multiple wide area wireless links
(e.g., EVDO, HSDPA, WiMax)
VanProxy
WiFi
Internet
1. Opportunistic erasure coding
2. Delay-based path selection
LanProxy
(b)
Figure 1: Two architectures for providing connectivity on-board buses. (a) Current practice. (b) PluriBus.

by striping data across available links based on the estimated delivery delay along each link [9]. We estimate delivery delay by estimating queue length and propagation delay, and send each packet on the link that offers the least delivery delay at that time. The striping decision for each packet is taken independently. In other words, PluriBus does not use a slower path until the queues on a faster path increase its delay to match the slower path, thus minimizing the average delay for data packets.

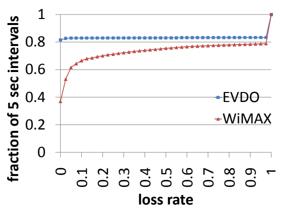
PluriBus has been deployed on two of Microsoft's campus buses for two months. Each bus is equipped with two WWAN links, one EVDO and one WiMax. We are currently working with Microsoft IT to put PluriBus in operational use.

We evaluate PluriBus using our deployment as well as emulator-based controlled experiments. In our deployment, it reduces the median flow completion time for a realistic workload by a factor of 2.5, compared to an existing method for spreading traffic across multiple links. We also study the opportunistic erasure coding and delay-based striping individually and find that they contribute roughly equally to the overall gain.

This paper makes the following contributions. First, it develops a transmission strategy that opportunistically consumes spare capacity. This strategy can be used to transfer any kinds of low-priority information (e.g., logs) in a way that does not hurt higher-priority data. Second, it presents a new rateless code, called Evolution codes. This code can be used in any setting where fast, partial recovery of data is more important than efficient, full recovery. Third, it shows that the combination of opportunistic erasure coding and delay-based striping, provides a high-performance method for bonding paths that have disparate delays, capacities and loss rates. Most existing works on bonding paths assume identical links, identical delays, or ignore losses [10, 30, 14, 9].

2. THE VEHICULAR ENVIRONMENT

In this section, we briefly characterize the paths and workload of our target environment. We show that the network paths are highly lossy. They also have high delays, a significant portion of which is inside the providers' network itself. The workload is dominated by short flows



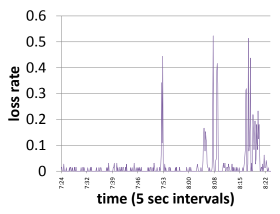


Figure 2: (a) The CDF of loss rate for paths to the buses. (b) A chosen hour-long window that shows the temporal behavior of loss rate for WiMax.

and is highly bursty. These characteristics motivate our solution, which is presented in the following sections.

2.1 Our Testbed

Our testbed consists of two buses that ply around the Microsoft campus. We equipped each with a desktop computer. The buses operate approximately from 7 AM to 7 PM on weekdays. The on-board computers are equipped with an 1xEVDO (Rev. A) NIC on the Sprint network and a WiMax modem (based on the draft standard) on the Clearwire network.[1]

2.2 Network Path Characteristics

We characterize path quality by sending packets between the bus and a computer connected to the wired Internet. Unless otherwise specified, a packet is sent along each provider in each direction every 100 ms and the analysis is based on two weeks of data.

2.2.1 Paths are highly lossy

Figure 2(a) shows the CDF of loss rates, averaged over 5 seconds, from the wired host to the buses. The reverse direction has a similar behavior. For EVDO, 20% of the intervals have a non-zero loss rate, while for WiMax, this number is 60%. For both, a significant fraction of the intervals have a very high loss rate. Figure 2(b) shows one handpicked hour-long window with a particularly bad loss behavior. These results agree with earlier measurements from moving vehicles in different countries [27, 13]. Such lossy behavior can hurt many applications, especially those that use TCP.

[1]We also experimented with an HSDPA card from AT&T. Its performance is qualitatively similar to the EVDO link.

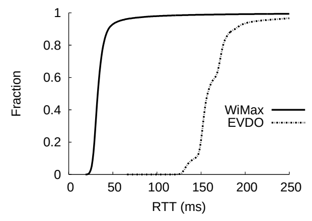


Figure 3: The CDF of path RTTs.

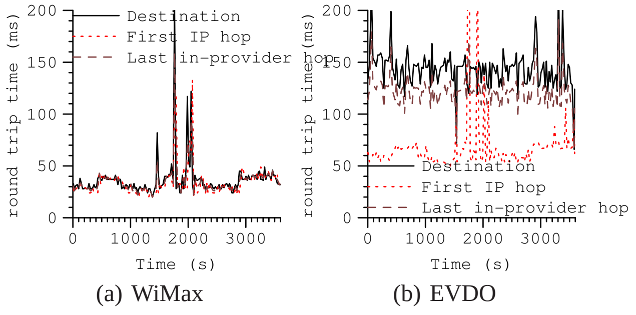


Figure 4: Breakdown of path RTTs.

2.2.2 *Paths have high and disparate delays*

Figure 3 shows the CDF of round trip time (RTT) for each provider. The median RTTs for both are rather high – roughly 40 ms for WiMax and 150 ms for EVDO – even though the path end points are in the same city.

To uncover where the time is spent, we run traceroute from the bus to the wide-area host. From this data, we extract the RTTs to the destination, to the first IP hop, and to the last hop in the wireless provider's network. The last hop is inferred using DNS names of the routers [31].

Figure 4 shows the results for an hour-long window. Surprisingly, a third of the delay for EVDO and nearly all of it for WiMax is to the first IP hop. For both, nearly all of the delay is inside the provider network. As we shall see later, this observation has implications for how losses can be masked in this environment.

Additionally, the factor of three difference in the RTT of the two providers implies that simple packet striping schemes like round robin will perform poorly. They will significantly reorder packets and unnecessarily delay packets along the longer path even though a shorter path exists in the system. Note that sending all the data on the shorter path is not possible due to capacity constraints. Using two links from the same provider would remove the delay disparity, but it would also reduce reliability, as the performance of the links will be highly correlated [27].

2.3 Workload Characteristics

To get insight into the workload in our target environment, we collect traffic logs from commuter buses that carry Microsoft employees to and from work. These buses have the setup shown in Figure 1(a), with a Sprint-based EVDO Rev. A NIC in the proxy device. We sniffed the intra-bus WiFi network on 11 different days to capture packets that are sent and received by the riders. The average number of active clients per trip is around four.

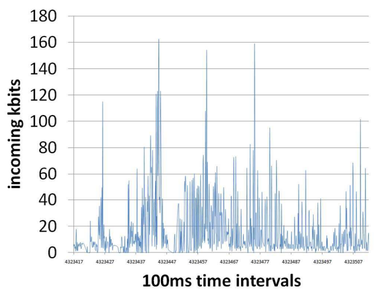


Figure 5: Traffic arriving for the clients from the Internet during 100 seconds

The essential characteristics of this workload are similar to those in many other environments. Traffic is dominated by short TCP flows, which are especially vulnerable to packet loss. It is also highly bursty, as illustrated by the example 100-second period shown in Figure 5. The average load over the entire measurement period is quite low, only 86 Kbps. However, short-term load on the link can be as high as 1.5 Mbps, which is roughly the saturation load and suggests capacity limitations. Burstiness makes it hard to accurately predict short-term traffic intensity or leftover capacity. As we discuss later, it also makes it hard to use existing erasure coding methods because of the difficulty in estimating how much redundancy can be added without overloading the path.

3. OVERVIEW OF PluriBus

Given the poor quality of wide-area wireless paths from moving vehicles, how can we best improve application performance? We could urge the carriers to improve the underlying connectivity. This would require significant investment towards improving coverage and handoffs. It is also a long-term proposition and does not help with the performance and growth of these networks today.

We instead take the approach of building a high-performance system on top of multiple unreliable links. Multiple links can help improve system reliability [27] and provide additional capacity to handle bursts in the load.

The PluriBus architecture is shown in Figure 1(b). The VanProxy is equipped with multiple wireless links. All packets are relayed through LanProxy, which is located on the wired Internet. Such relaying allows us to mask packet losses on the wireless links, which is not possible if the VanProxy directly communicated with remote computers. It also allows us to stripe packets across links in a fine-grained manner. Otherwise, striping must occur at least at the level of individual connections, which performs poorly (§6.3).

It may seem that relaying via LanProxy will increase end-to-end latency. However, because of the high delay inside wireless carrier networks, any increase should be small if the LanProxy is deployed within the same city. Internet path latencies within a city tend to be small [31]. Interestingly, relaying through our deployed LanProxy actually reduces latency to most destination because of significant Detour effects [29].

Within the context of this architecture, we can now concisely state the problem we want to address. We are given one or more paths between the two proxies. Each path uses a different WWAN link. Paths are lossy with time-varying loss rate. Different paths have different capacities and delays. The incoming data at each proxy is bursty and arrives at an unknown and time-varying rate.

We assume that the WWAN links are the bottlenecks. We further assume that the MACs of these links isolates transmitters such that aggressive usage by one does not hurt others. 3G and WiMax MACs achieve this by tightly coordinating medium access at the basestation.

Our goal is to deliver data from one proxy to the other in a manner that minimizes completion times for interactive traffic such as Web transfers. Per this goal, we aim to minimize loss and delay experienced by end hosts, the two factors that impact completion time.

PluriBus uses opportunistic erasure coding to leverage spare path capacity to mask losses from end hosts, and it uses delay-based striping to transmit each packet on the path that currently offers the least delay.

1. Opportunistic erasure coding: We can mask losses either by retransmitting lost packets based on feedback from the other proxy or by proactively sending erasure coded packets. The former is slow in our setting because it takes at least 1.5 times the inter-proxy RTT, which tends to be high. Additionally, it may not hide many losses if end hosts detect loss within one end-to-end RTT (e.g., using TCP), as inter-proxy RTT can be major portion of the end-to-end RTT.

Erasure coding is thus a better fit for our setting. We desire two properties from it, which dictate when and how many coded packets are sent, and what code generates them. First, coded packets should interfere minimally with data packets, while providing as much protection as possible. Coded packets interfere when data packets are queued behind them at the bottleneck queue because this amounts to them stealing valuable capacity from the data packets.

None of the existing methods, such as Maelstrom [2] or CORE [21], fulfill our first property. These methods generate a fixed number of coded packets for a given set of data packets, and the coded packets are sent regardless of current state of the queue. If this fixed overhead is low, they do not provide sufficient protection even though there may be excess capacity in the system. If it is high, they hurt application throughput by stealing capacity from data packets. Given the bursty nature of incoming traffic, tuning overhead to match spare capacity at short time scales is difficult.

In PluriBus, we send coded packets opportunistically, that is, when and only when there is instantaneous spare capacity in the system. We judge the availability of spare capacity by estimating the length of the bottleneck queue. This way, coded packets defer to data packets and delay them by at most one packet, and they provide as much protection as the amount of spare capacity allows.

The second property is that the code should not rely on the receiver getting some minimum number of coded packets. With bursty incoming traffic and thus available capacity, such a requirement is hard to meet. Conventional erasure codes, whether rateless (e.g., LT [22]) or not (e.g., Reed-Solomon [26]), do not have this property. They are designed to minimize the number of packets needed at the receiver to fully recover all data packets. But they recover very little if fewer that this number of packets are received [28].

For PluriBus, we design *Evolution* codes which maximize the expected number of data packets recovered using each coded packet. Thus, instead of aiming for efficient, full recovery, we aim for greedy, partial recovery based on whatever the receiver manages to get.

Additionally, the set of data packets we encode over is a moving window because we consider only packets that arrived at the sending proxy in the last inter-proxy RTT. For older packets, the end hosts may have already detected loss and retransmitted. Evolution codes naturally apply to moving windows, which is not the case for many of the existing codes.

Logically, opportunistic erasure coding uses all spare capacity along a path, to maximize the level of protection. In practice, however, path are not always fully used (§6.2.3) because coded packets are sent only if new data packets arrived within the last inter-proxy RTT. Nevertheless, we do use the paths aggressively. We discuss the implications of this behavior in §8.

Finally, observe that our scheme strictly prioritizes data packets over coded packets because we send coded packets only when the queue is empty. It is based on a packet-level view and maximizes the rate at which new data reaches the receiver; while a data packet delivers one new data packet at the receiver, coded packets deliver less on average. If packets have complex dependencies, this prioritization may not be optimal. For example, consider the case where the utility of a newly arrived data packet depends on earlier data packets already being at the end host (e.g., as in MPEG encoding). Here, the right strategy might be to transmit coded packets that try to recover the earlier packets before the new data packet is transmitted. We choose not to optimize for such dependencies, to keep our design simple. Also note that since our proxies multiplex several simultaneous connections, many packets would in fact be independent.

2. Delay-based path selection: PluriBus sends each data packet along the path that is likely to deliver it first [9], which we judge using estimates of queue length and propagation delay. It continues to send traffic along the fastest path until queue buildup brings its delay up to the level of the next fastest path, and so on. This method naturally generalizes striping mechanisms such as round robin to the case of paths with different delays and capacities. It minimizes the average delay experienced by traffic and also makes it less likely for a later packet to arrive before a previously sent packet. Variations and mis-estimations of path delay can still lead to some reordering, which we handle using a small sequencing buffer.

Observe that our striping minimizes average packet-level delay, which may not translate to minimum average completion time. A scheme that prioritizes packets from short connections over those from long ones may result in lower average completion times. We choose a striping mechanism that is independent of connection sizes due to its simplicity.

To summarize, PluriBus transmits data packets as soon as they arrive, along the path deemed to have the least delay. It sends coded packets along a path only when its estimated queue length is zero. The contents of a coded packet are determined using Evolution codes.

4. DESIGN OF PluriBus

We now describe our design in more detail. We start by describing Evolution codes. Next, we describe how we estimate the queue length along a path. Finally, we describe how we select the path that has the least delay.

For the purpose of this section, the terms sender and receiver refer to the two proxies. Identical algorithms run in each direction.

4.1 Evolution codes

Evolution codes aim for greedy, partial recovery, by maximizing the expected number of packets that will be recovered with each coded packet. At any given instant, the sender encodes over a set of data packets W that were sent within the previous RTT. Let r be the fraction of the W packets (but not which exact packets) that has been successfully recovered by the receiver. For tractability, we assume that each packet in W has the same probability, equal to r, of being present at the receiver. In practice, the probabilities of different packets may differ. We describe later how the sender estimates r based on past transmissions of data and coded packets.

Given current values of W and r, how should the next packet be coded? To keep encoding and decoding operations simple, we only create coded packets by XOR-ing data packets together. Because of the assumption that all packets have the same probability of being there at the receiver, the question boils downs to how many packets should be XOR'd. A simple analysis yields the optimal number of packets that must be included. It assumes that coded packets that could not be immediately decoded at the receiver are discarded, and thus a coded packet can recover at most one data packet.

Suppose the sender XORs x ($1 \leq x \leq |W|$) data packets. The probability that this coded packet will yield a previously missing data packet at the receiver equals the probability that exactly one out of the x packets is missing. That is, the expected yield $Y(x)$ of this packet is:

$$Y(x) = x \times (1 - r) \times r^{x-1} \tag{1}$$

$Y(x)$ is maximized for $x = \frac{-1}{ln(r)}$. This result can be intuitively explained. If the number of data packets already at the receiver is low, the coded packet should contain few data packets. For instance, if more than half of the packet are missing, the best strategy is to code only one packet at a time (i.e., send duplicate packets); coding even two is likely futile as the chance of both being absent, and hence of nothing being recovered, is high. Conversely, if many packets are present at the receiver, encoding a higher number of packets is the more efficient way to recover missing data.

Thus, in PluriBus, the sender selects $\max(1, \lfloor\frac{-1}{ln(r)}\rfloor)$ data packets at random to XOR. We round down because including fewer data packets is safer than including more. Further, if $|W| > 1$ and $\lfloor\frac{-1}{ln(r)}\rfloor \geq |W|$, we XOR only $|W|-1$ packets. We never XOR all W of packets because of a subtle corner case that arises when the window of packets is not changing and more than one data packet is missing at the receiver, but the sender estimates that fewer data packets are missing. In this case, XOR-ing all W packets leads to repeated transmissions of the same coded packet that cannot recover anything new.

Updating W and r: The sender updates the set of packets W and estimate the fraction r as follows.

i) When a new data packet is sent, it is first included in W and then r is updated to reflect the probability that the new packet is received:

$$r \leftarrow \frac{(|W| - 1) \times r + (1 - p)}{|W|}$$

p is a rough estimate of the loss rate of the path along which the packet is sent. Receivers estimate p using an exponential averaging of past behavior and periodically inform the sender of the current estimate. In §6.2.2, we show that PluriBus gets a fairly accurate estimate of loss rate and Evolution codes are robust to small inaccuracies.

ii) When a coded packet, formed by XOR'ing x data packets, is sent, W does not change, and r is updated to reflect the probability that the coded packet is received, and yielded a new packet:

$$r \leftarrow \frac{|W| \times r + (1 - p) \times Y(x)}{|W|}$$

where $Y(x)$ is defined in Eq. 1.

$iii)$ When the receiver returns the highest sequence number that it has received – this information is embedded in packets flowing in the other direction (§5) – packets with lower or equal sequence numbers are removed from W. This step ensures that the sender encodes only over roughly one round trip of data. We set r to the average path loss rate observed by the data packets that remain in W. This step may underestimate the fraction of packets at the receiver because it does not account for the fact that some of the packets in W may have been recovered using previously sent coded packets. We find that it does not hurt in practice and yields better performance than leaving r unchanged which tends to overestimate the fraction of packets currently at the receiver.

We now reveal the rationale for the name "Evolution." In this code, the complexity of the coded packets, i.e., the number of included data packets, evolves with link conditions, window of data packets, and past history of coded packets. The complexity of the coded packets increases as more coded packets are generated and decreases when new data is included in the window.

Note the following regarding Evolution codes. First, they explicitly consider the impact of loss rate. Existing codes do not, which is striking given their goal of combating packet loss. Second, because of their focus on greedy, partial recovery, Evolution codes are less efficient for full recovery. The receiver needs more packets to recover all data packets. Our simulations show that this inefficiency is 15-20%. Third, our decoding algorithm discards any packets that we cannot immediately decode. This simplifies implementation but sacrifices some performance. Our simulations show that the penalty is negligible (1-2%) because we send simple packets before complex packets. We omit both sets of simulations results due to space constraints.

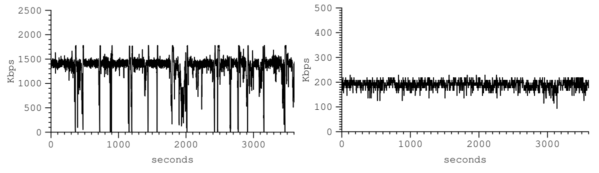
4.2 Estimating Queue Length

We maintain an estimate of queue length along a path in terms of the *time* required for the bottleneck queue to fully drain. It is zero initially and is updated after sending a packet:

$$Q \leftarrow max(0, Q - TimeSinceLastUpdate) + \frac{PacketSize}{PathCapacity}$$

PathCapacity refers to the capacity of the path, which we estimate using a simple method described below. The capacity of a path is the rate at which packets drain from queue at the bottleneck link. It is different from throughput, which refers to the rate at which packets reach the receiver. The two are equal in the absence of losses.

The WWAN MACs control media usage by individual transmitters, making it easier to estimate the capacity of such links compared to CSMA-based WiFi links. As an example, Figure 6 shows the throughput of WiMax paths in the downlink and uplink direction for one-hour windows in which we generate traffic at 2 Mbps in each direction. We see roughly stable peak throughputs of 1500 and 200 Kbps, which correspond to their capacity. An analysis of incoming sequence numbers confirms that throughput dips are due to packet losses and not slowdowns in queue drain rate.



(a) Downlink (b) Uplink

Figure 6: The downlink and uplink throughput of WiMax paths. The y-axis range of the two graphs is different.

While the behavior above may suggest that we could simply configure path capacities, we include an estimation component to be robust to variations that may occur as the vehicle traverses through different regions. We use the insight behind recent bandwidth measurement tools [16, 17]: if the sender sends a train of packets faster than the path capacity, the receive rate corresponds to the path capacity. Instead of using separate traffic to measure capacity, we leverage the burstiness of data traffic and the capacity-filling nature of our coding method to frequently create packet trains with a rate faster than path capacity.

We bootstrap the proxies with expected capacity of the paths. The receiver then watches for changes. It measures the rate of incoming packets directly and computes the sending rate using timestamps that the sender embeds in each packet. The two rates are computed over a fixed time interval (500 ms in our experiments). The capacity estimate is updated based on intervals in which the sending rate is higher than the current capacity estimate. If the receive rate is higher than the current capacity estimate for three such consecutive intervals, the capacity estimate is increased to the average of current estimate and the median of the three receive rates. If the receive rate is lower for three consecutive intervals, the capacity estimate is decreased to the average of the current estimate and the median of the three receive rates. Changes in capacity estimate are communicated to the sender.

Errors in capacity estimate can lead to errors in the queue length estimate. In theory, this error can grow unboundedly. In practice, we are aided by periods where little or no data is transmitted on the path, which are common with current workloads. These periods reset our estimate to its correct value of zero. While we cannot directly measure the accuracy of our queue length estimate, we show in §6.3 that our path delay estimate, which is based in part on this estimate, is fairly accurate.

4.3 Identifying Minimum Delay Path

To send data packets, PluriBus needs to estimate the current delay along the path. A simple method is to use the running average of one-way delays observed by recent packets, based on feedback from the receiver. However, we find that this method is quite inaccurate because of feedback delay and because it cannot capture with precision short time scale processes such as queue build-up along the path (§6.3). Capturing such processes is important to consistently pick paths with the minimum delay.

Our estimate of path delay is based on $i)$ transmission time, which primarily depends on path capacity; $ii)$ queue length; and $iii)$ propagation delay. We described above how we estimate the first two. Measuring propagation delay requires finely synchronized clocks at the two ends, which may not be always available. We skirt this difficulty by observing that we can identify the faster path even if we only computed the propagation delay plus a constant that is unknown but same across all paths. This constant happens to be the current clock skew between the two proxies.

Let the propagation delay of a path be d and the (unknown) skew between the two proxy clocks be δ. A packet that is sent by the sender along the path at local time s will be received by the receiver at local time

$$r = s + \delta + d + QueueLength + \frac{PacketSize}{PathCapacity}$$

If the packet is sent when the queue length is zero:

$$d + \delta = r - s - \frac{PacketSize}{PathCapacity}$$

We can thus compute propagation delay plus skew using local timestamps of packets that see an empty queue.

To enable the above estimate, senders embed their timestamps in the transmitted packets. The receivers keep a running exponential average of $r - s - \frac{PacketSize}{PathCapacity}$ for each path, which corresponds to $(d + \delta)$. Only packets that are likely to have sampled an empty queue are used for computing the average. Packets that get queued at bottleneck link are likely to arrive roughly $\frac{PacketSize}{PathCapacity}$ time units after the previous packet. We use in our estimates packets that arrive at least twice that much time after the previous packet. The running average is periodically reported by the receiver to the sender.

It is now straightforward for the sender to compute the path that is likely deliver the packet first. This path is the one with the minimum value of $\frac{PacketSize}{PathCapacity} + QueueLength + (d + \delta)$. This sum is in fact an estimate of the local time at the receiver when the packet will be delivered. We show in §6.3 that despite the approximations in the computation, our estimates are fairly accurate.

5. IMPLEMENTATION

We now describe our implementation of PluriBus. When the VanProxy boots, it uses one of its wide-area interfaces to inform the LanProxy of the current IP addresses of its wireless interfaces. The LanProxy sends configuration information to the VanProxy, including the IP address range that should be used for the clients. Clients use DHCP to get their configuration information from the VanProxy.

The two proxies essentially create a bridge by tunneling packets over the paths between them. When the VanProxy receives an IP packet from a client for a remote computer, it encapsulates this packet and a custom header (described below) into a UDP packet.[2] It sends the encapsulated packet to the LanProxy which decapsulates and relays the original IP packet to the remote computer.

In the reverse direction, packets from the remote computer for a vehicular client reach the LanProxy. The packet's destination IP address is used by the LanProxy to determine the target VanProxy. (The LanProxy may be serving multiple VanProxies.) After encapsulation, the packet is relayed to the VanProxy which decapsulates and sends it to the client.

PluriBus uses multiple sequence number spaces. One space is used for data packets that arrive at the proxy to be sent to the other proxy. These data-level sequence numbers let the receiver uniquely identify data packets and their relative order. There is also a path-level sequence number space for packets transmitted along a path. This space helps the receiver estimate various path properties, such as loss rate.

Each PluriBus proxy caches incoming data packets for a brief window of time so that it can decode coded packets. It also has a sequencing buffer to order received data packets. When a decoded or received data packet has a sequence number higher than one plus the highest sequence number relayed, it is stored in this buffer. It is relayed as soon as the missing data packets are received or decoded or when a threshold amount of time, set to 50 ms in our experiments, elapses.

PluriBus packets have a header with five fields: message type, timestamp (in milliseconds), data- and path-level sequence numbers, and the highest data-level sequence number received by the sender. The message type is a 1-byte field that differentiates data, coded, and control packets. Other fields are two bytes each. Coded packets also contain a 4-byte bitmap that encodes which data packets are contained in it, relative to the data-level sequence number. Control packets are used by the proxies to exchange configuration information, report on wide-area address changes at the the VanProxy, and properties of incoming paths.

The PluriBus header and the encapsulation lowers the

[2]We do not use the lower-overhead IP-in-IP encapsulation because such packets are filtered by our wireless providers. In fact, our UDP packets have to masquerade as DNS responses to get past the firewalls.

effective link MTU by 41 bytes. To minimize the chances of fragmentation, we inform the clients of the lower MTU via the Interface MTU option of DHCP. Some clients inform their wide area peers of their MTU during TCP connection establishment, via the MSS option of the SYN packet. For other clients, we are experimenting with modifying the MSS option of TCP SYNs as they traverse the VanProxy. With these changes, only large UDP packets destined for the clients will be fragmented; such packets constitute a miniscule fraction in our traces. VPNs face a similar fragmentation issue.

6. EVALUATION

Along with studying the overall performance of PluriBus (§6.1), we study in detail the behavior of opportunistic erasure coding (§6.2) and of delay-based striping (§6.3).

Our primary experimental platform is the deployment on buses that operate regularly on Microsoft campus (§2.1). This platform offers a real environment in which we can study the performance of PluriBus and competing policies. Here, we do not control anything (e.g., path loss and delay, and their variation) except the workload. To isolate individual factors, we complement this platform with controlled experiments using a network emulator. To avoid confusion, our result graphs clearly indicate the experimental platform upon which they are based.

Workload: For the experiments presented in this paper, we generate realistic, synthetic workloads from the traces described in §2.3. We first process the traces to obtain distributions of connection sizes and inter-arrival times, where a connection is the standard 5-tuple. The synthetic workload is based on these distributions of connection sizes and inter-arrival times [11]. The average demand of this workload is 86 Kbps but it is highly bursty.

To study the performance of PluriBus as a function of load, we synthesize scale versions of the workload by scaling the inter-arrival times. To scale by a factor of two, we draw inter-arrival times from a distribution in which all inter-arrival times are half of the original values, while retaining the same connection size distribution. Our workload synthesis method does not capture many details, but we believe it captures to a first order the characteristics that are important for our evaluation. As per the metrics that we use below, the performance of a synthetic workload scaled by a factor of 1 is similar to an exact replay of connection size and arrival times.

To verify if our conclusions apply broadly, we also considered other workloads. These include controlled workloads composed of a fixed number of TCP connections and those generated by Surge [4], a synthetic Web workload generator. The results are qualitatively similar to those below. We omit details due to lack of space.

Performance measure: We use connection completion time as the primary measure of performance. This is of direct interest to interactive traffic such as short Web

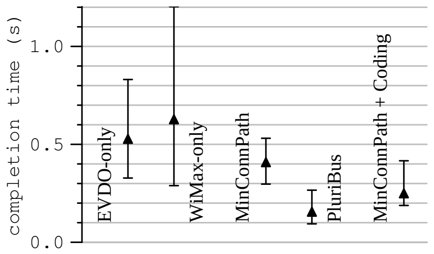


Figure 7: Performance of various systems for a workload based on our traces. [Deployment]

transfers that dominates in our environment. We use the median as the representative measure and the inter-quartile range (25-75%) to capture the observed spread.

Baseline for performance comparison: Connection-level striping is the state-of-the-art when multiple uplinks are used in a vehicular setting [27]. All packets of a connection traverse the same path, and no special loss recovery (beyond end-host TCP) is performed. Of the several possible connection-level striping policies, we use a policy called *MinConnPath*. It maps a new connection to the path with the minimum number of active connections. Active connections are expired when they no packet is received for them for 30 seconds.

MinConnPath performs better than other connection-level policies that we also experimented with. Round robin does worse because it does not consider current path load while mapping new connections. Mapping new connections to links that currently carry less load in terms of bytes per second performs worse as well. Lower load on a link can stem from its poor performance – because the traffic is responsive – and this policy ends up mapping more connections to the poorer link.

6.1 Overall performance

Figure 7 shows the performance of various means of transferring data to and from the bus. For each method, the graph shows the median and the inter-quartile range of the connection completion times. These results are based on our deployment on-board buses. Each method ran for at least two days during which time it complete tens of thousands of connections.

The first two bars show the performance of the system when only one of the wireless uplinks is used directly, as is the norm today. The median completion time of EVDO is 500 ms and of WiMax is 700 ms. WiMax offers lower performance because of its higher loss rate, even though it has a much lower round trip time. The observed average loss rate in this experiment is 5% for WiMax and under 1% for EVDO. The higher loss rate of WiMax also explains its larger inter-quartile spread. The completion time increases significantly for connections that happen to suffer a loss.

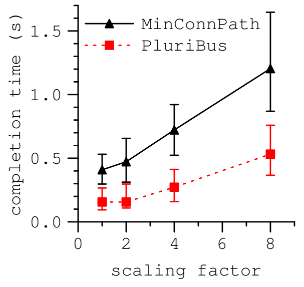


Figure 8: Performance of *MinConnPath* and PluriBus as a function of load. [Deployment]

The third bar shows the performance when both links are employed simultaneously using *MinConnPath*. We see that *MinConnPath* reduces the median completion time to 400 ms, by spreading load across both paths. We find that it sends roughly equal amount of data along each path on average. The spread in completion time is lower because a smaller fraction of connections suffer losses compared to the WiMax-only case and queues (due to bursts) build less often due to additional capacity.

The fourth bar shows the performance of PluriBus. We see that PluriBus performs significantly better than *MinConnPath*. Its median completion time is 150 ms, which represents a reduction factor of 2.5 over *MinConnPath*. We also find that PluriBus reduces the loss rate seen by end hosts to almost zero. This loss rate is roughly 2% with *MinConnPath*.

The better performance of PluriBus stems from the combination of opportunistic erasure coding and per-packet delay-based striping. To tease apart their individual contributions, we added opportunistic erasure coding to *MinConnPath*. The fifth bar in Figure 7 shows that the median completion time of *MinConnPath* + *Coding* is 250 ms. Thus, the addition of coding improves performance by a factor of 1.6 (400/250). The median completion time of PluriBus is even lower. This difference, which amounts to an improvement factor of 1.6 (250/150) can be attributed to the delay-based striping of PluriBus.

We now study the performance of PluriBus under higher load. Figure 8 plots the median and inter-quartile range for flow completion time as a function of the scaling factor used for the synthetic workload. Each data point is based on at least two days of data. We see that the performance advantage of PluriBus persists even when the workload is scaled by a factor of eight. Even at such high load levels, there is ample instantaneous spare capacity for PluriBus to mask losses and improve performance.

Figure 8 also shows that the performance of PluriBus with a scaling factor of eight matches that of *MinConnPath* with a scaling factor of one. This implies that if the desired median completion time is 400 ms, PluriBus can support eight times as much load for the same two WWAN links.

Having studied the performance of PluriBus as a whole, we focus next on studying in more detail the effectiveness of its individual components.

6.2 Opportunistic erasure coding

We start with opportunistic erasure coding. We compare it to other potential methods for masking packet loss, evaluate the inaccuracy of loss rate estimation in vehicular environments and its performance impact, and quantify the impact of aggressive coding.

6.2.1 Benefit relative to other methods

The erasure coding method used in PluriBus can be thought of as two separate mechanisms. The first mechanism, opportunistic transmission, controls *when* coded packets are sent. It sends coded packets only when spare capacity is available. The second mechanism, Evolution coding, controls *what* coded packets are sent. It stresses partial recovery by explicitly considering information already present at the receiver. To better understand the contribution of these two mechanisms, we compare our method to two alternative methods for erasure coding.

The first method is based on fixed overhead codes such as Reed-Solomon, for which the fraction of coded to data packets is independent of prevailing workload and available capacity. This method has neither opportunistic transmissions nor a code meant for partial recovery.

To implement a fixed-overhead code with K% redundancy, we send a coded packet after every $\frac{100}{K}$-th pure packet. Each coded packet codes over packets in the current unacknowledged window since the last coded packet. Thus, when $K = 100$, every other packet is coded and carries the previously sent pure packet (i.e. every packet is simply sent twice); when $K = 10$, every 11^{th} packet is the XOR of previous 10 pure packets that still remain in the unacknowledged window. This method is a simple version of fixed-overhead codes and is similar to $(K, 1)$ Maelstrom code [2].

The second method is based on rateless codes, such as LT, that can be adapted for opportunistic transmissions because of they can generate on-demand as many coded packets as needed. The coded packets generated by this method are not based on what might be already present at the receiver.

Our adaptation of rateless codes sends coded packets opportunistically, like PluriBus. The degree distribution of coded packet is decided based on the current size of the unacknowledged window. We use the Robust Soliton degree distribution, which is what LT codes use. This distribution uses two input parameters, c and d, which we set to the commonly used values of 0.9 and 0.1. Because of small window sizes that dominate our environment, other values yield similar results.

In addition to erasure coding, we also consider a system that retransmits lost packets based on receiver feed-

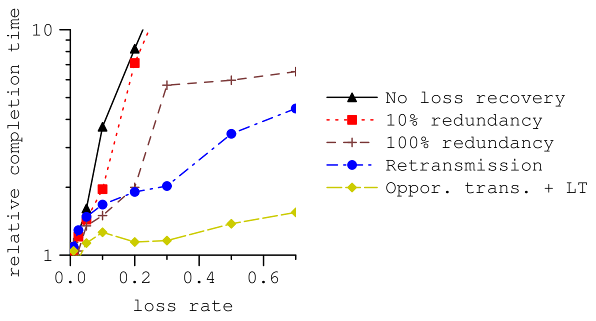


Figure 9: Performance of different loss protection methods relative to the coding method of PluriBus. [Emulation]

back. This is the simplest loss recovery method and studying it shows the value of erasure coding in PluriBus.

To evaluate these alternate methods, we perform controlled experiments using the network emulator. We configure only one link between VanProxy and LanProxy, to exclude the impact of striping. The link has a one-way delay of 75 ms and capacity of 1.5 Mbps. The loss rate on the link is varied from 1% to 70%. We show results from using the Bernoulli loss model, in which each packet has the same loss probability. We omit result using more sophisticated loss models such as Gilbert-Elliot that induce more bursty losses; they are qualitatively consistent with those below.

Figure 9 shows the results as a function of the loss rate. For each method, we plot its median completion time divided by the median completion of PluriBus. K% redundancy curves correspond to fixed-overhead schemes. All y-values in this graph are greater than one because all other methods performs worse than PluriBus.

We draw several conclusions from this graph. First, as expected, some form of loss protection significantly boosts performance. For instance, at 10% loss rate the relative completion time without any loss protection is at least twice that of any method of loss protection.

Second, PluriBus does better than retransmission-based loss recovery. As mentioned earlier, this is because its erasure coding is able to recover faster from losses, which improves performance and also makes it less likely to conflict with recovery efforts of the end hosts.

Third, PluriBus outperforms fixed-overhead method, at both levels of redundancy and at all loss rates. This is surprising at least at some data points. For example, at 10% channel loss rate, 100% redundancy sends each packet twice and recovers most channel losses. Its performance is nevertheless poor. The problem is that the workload is bursty, and adding a fixed overhead without regard to instantaneous load results queue buildup. On the other hand, by using opportunistic transmissions, PluriBus avoids these slowdowns, even though it sends more coded packets on average.

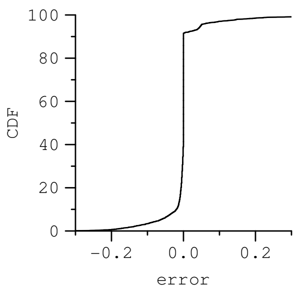


Figure 10: Accuracy of loss rate estimates in PluriBus. [Deployment]

Fourth, PluriBus performs better than the combination of opportunistic transmissions and LT coding by about 10-50% In this case, the advantage of PluriBus lies in it using Evolution codes to guide the complexity of coded packets based on an estimate of what is present at the receiver. LT do not account for information at the receiver and thus sometimes either send an overly complex coded packet (i.e. too many packets XOR'd together) that are less likely to be decodable or send very simple coded packets that are less likely to include new information.

Collectively, these results also indicate that opportunistic transmissions are more valuable than Evolution coding. The performance differences between Evolution and LT coding (with opportunistic transmissions) are smaller compared to those between settings with and without opportunistic transmissions. This is result of the fact that window sizes over which coding is done is usually small in this environment. In separate experiments, we find that the relative advantage of Evolution codes increases as window sizes increase.

6.2.2 *Accuracy of loss rate estimate*

Evolution codes take loss rate into account while generating coded packets. Given the dynamics of the vehicular environment, loss rate maybe hard to estimate. Figure 10 shows that we manage to get a pretty accurate estimate of loss rate in our deployment. It plots the difference in the loss rate for the next twenty packets minus the current running average of the loss rate that we use to predict future loss rate. Over 90% of the time, our estimate is within $\pm 10\%$.

Despite the accuracy of loss rates estimates in our deployment, it is important to understand the performance impact of any inaccuracies. To study this, we use the same emulator setup as above but force the proxies to use an inaccurate value of loss rate.

Figure 11 shows the results, when we program the proxies to use loss rate offsets of ± 0.1 and ± 0.2 off of the actual loss rate on the emulated link. Positive offsets represent overestimation of the loss rate and negative offsets represent underestimation. In the graph, the x-axis corresponds to the actual loss rate and individual bars correspond to different offsets. There is no bar for the

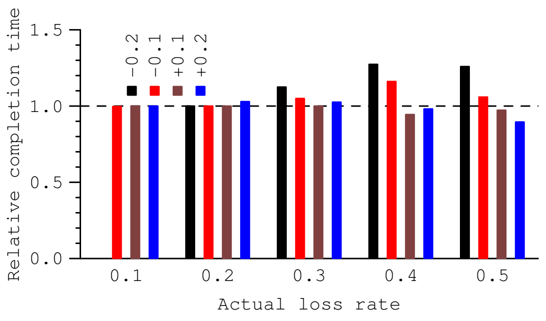


Figure 11: Impact of inaccuracy in loss rate estimate in PluriBus. [Emulation]

case of actual loss rate of 0.1 and an offset of -0.2. The y-axis plots the median completion time relative to the case where we do not program a loss rate offset.

We see that for low actual loss rates, inaccuracies in loss rate estimates have little impact on performance. At higher loss rates, which are uncommon in our environment, there is some performance degradation with underestimation. This degradation is only about 20% if the underestimation is by 0.2.

Interestingly, at high loss rates, performance improves if loss rates are overestimated. This stems from our strategy of greedily maximizing the expected yield of each coded packet, ignoring possible optimizations over groups of coded packets. We do this for simplicity and also because we do not know at the time of transmission if more can be sent. The greedy strategy, however, can generate coded packets that are sometimes too complex to be decodable at the other end. At high loss rates, groups of simpler packets can outperform groups of coded packets generated by PluriBus. In the experiment, overestimating loss rate helps because it leads to generating simpler packets. We are currently investigating means for extending Evolution codes to optimize over small groups of coded packets.

6.2.3 *Impact of aggressive coding*

A potential negative side-effect of our strategy to aggressively send coded packets is slowdown for data traffic in environments where the loss rate is low. However, we find that this does not occur because of our strategy of sending a coded packet only when the queue is estimated to be empty.

For this experiment, we consider a setting with no underlying loss because then coding brings no benefit and can only create overhead. Since losses are common in our deployment, we use emulation. We configure the emulated link between the two proxies to have zero loss rate, 150 ms round trip delay, and 1.5 Mbps capacity. We use scaled versions of our traces as workload and compare the performance of traffic with and without coding.

Figure 12(a) shows the results by plotting the median and inter-quartile flow completion time as a function of the scaling factor. We see that any slowdown with coding is minimal even at high load levels. Thus, our coding methodology does not hurt application performance in non-lossy environments and, as we have shown before, significantly boosts performance in lossy environments.

Finally, we study how much additional traffic is generated by our coding methodology. This factor might be of concern where wireless access is priced based on usage. In our own deployment, access fixed-priced, which is more common.

Figure 12(b) shows the results for experiments using our deployment. As expected, the fraction of coded packets declines as the workload intensifies because there are fewer opportunities to send coded packets. We can also see that while PluriBus logically fills the pipe, the actual amount of data sent is much lower because it codes over only data packets that arrive in the last RTT. At the scaling factor of 1, the average data traffic load is 86 Kbps. Given that roughly two-thirds of all packets are coded, the total load generated by PluriBus is roughly 258 Kbps, which is much lower than the combined capacity of our two links.

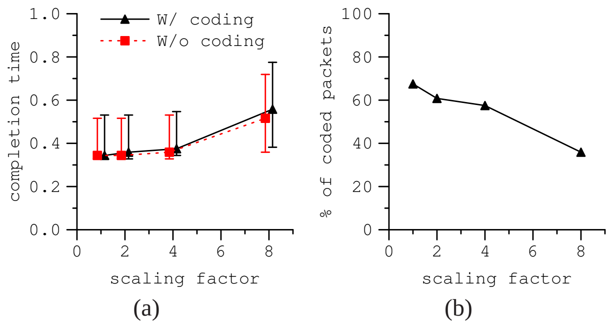


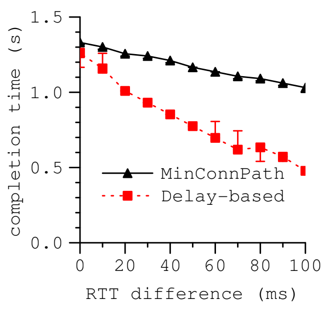
Figure 12: (a) Impact of coding on flow completion time. The lines connect the median and the error bars show the inter-quartile range. [Emulation] (b) Percentage of coded packets sent as a function of workload intensity. [Deployment]

6.3 Delay-based path selection

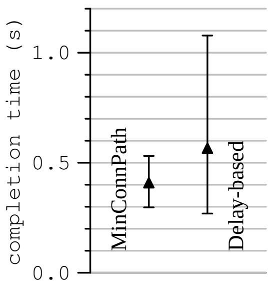
We now study in detail the behavior of delay-based path selection of PluriBus, without the use of coding.

6.3.1 *Benefit of fine-grained striping*

We first quantify the advantage of fine-grained, delay-based packet striping used by PluriBus by comparing it with *MinConnPath*. The workload consists of two simultaneous (but not synchronized) TCP flows. Each flow downloads 10 KB of data. A new flow starts when one terminates. This workload represents a particularly good case for *MinConnPath* because it is more likely to generate an even distribution. Even spreads are harder to achieve in realistic workloads because individual flows have different sizes. Any advantage of PluriBus in this two-TCP workload stems from its ability to stripe data at



(a)



(b)

Figure 13: Comparison of delay-based striping and *MinConnPath*. (a) As a function of delay difference, in environments without loss. [Emulation] (b) In a lossy environment, without coding. [Deployment]

the level of individual packets based on the current delay estimate of each link.

We configure the emulator with two links between the VanProxy and LanProxy. Each link has a capacity of 1.5 Mbps. The round trip propagation delay of one link is fixed to 150 ms and that of the other is varied from 150 ms down to 50 ms. Neither link has any inherent loss, and we configure our system to not using any coding. This allows us to isolate the impact of striping strategies.

Figure 13(a) shows the median and inter-quartile flow completion times for the two policies as a function of the difference in the emulated link RTTs.We see that as the difference in RTT increases the relative performance advantage of delay-based striping increases. In the extreme, when the difference in the RTT is 100 ms, which roughly corresponds to the difference in our deployment, delay-based striping halves the flow completion time.

Interestingly, even when the two links have equal delays and *MinConnPath* should lead to almost perfect distribution of connections across them, delay-based striping does slightly better. It does that by exploiting the short-term differences in the queue lengths along the two paths, which arise when the two TCP flows have different window sizes. Further experimentation shows that this advantage of delay-based striping becomes more prominent as transfer sizes increase because that creates bigger differences in the queues lengths.

These results show that delay-based striping outperforms *MinConnPath* in absence of losses. What happens when losses are present and no coding is used to mask them? In such a situation, the delay-based striping significantly *underperforms* the *MinConnPath* policy. Figure 13(b) shows this using our deployment and unscaled workload. The first bar, for *MinConnPath*, is same as that in Figure 7. The second bar is for delay-based striping with coding disabled. We see that it performs worse than *MinConnPath*, because the two links in our deployment have different loss rates. *MinConnPath* stripes at connection level; the connections that are mapped to the EVDO link suffer few losses and perform well. Delay-based striping, on the other hand, stripes at packet-level; many connections suffers from the frequent losses on the WiMax link.

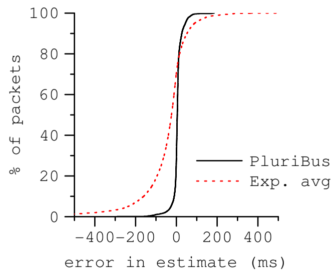


Figure 14: Error in estimated delay along a path. [Deployment]

These results imply that loss recovery is important for packet-level striping of PluriBus to be effective. A corollary is that if the load is so high that there is no spare capacity to send coded packets, PluriBus will do worse that *MinConnPath* if the paths have different loss rates. In our present deployment there is ample spare capacity even if we scale the workload eight-fold. We are currently investigating extensions by which PluriBus will gracefully switch to connection-level striping if there is not enough spare capacity to mask losses.

6.3.2 *Accuracy of path delay estimate*

Various factors in a real deployment, including estimates of path capacity, queue length, and propagation delay, impact the delay estimate of PluriBus. For good performance, the accuracy of this estimate is important. We evaluate accuracy by comparing the estimated delivery time at the sender to the actual delivery time at the receiver. This comparison is possible even with unsynchronized clocks because our estimate of propagation delay already includes the clock skew.

Figure 14 shows the error in our delay estimate for the deployment-based experiments of §6.1. It includes all load scaling factors. The curve labeled PluriBus shows that our estimate is highly accurate, with 80% of the packets arriving within 10 ms of the predicted time. This is encouraging, especially considering the inherent variability in the delay of wide-area wireless path (Figure 3).

One downside of any inaccuracy in the delay estimate is that packets might get reordered. Reordering translates to additional delay as the packet will be put in the sequencing buffer for the previous packet to arrive. We find that fewer than 5% of the packets arrive at the other end before a previously sent packet. 95% of such packets wait for less than 10 ms.

Finally, the curve marked “Exp. avg.” shows the error if we were to estimate path delay simply as an exponential average of observed delays, rather than the detailed accounting that we conduct based on estimated capacity and queue length. We see that it tends to significantly underestimate path delay. We also observe (but omit detailed result) that this translates to significant reordering and performance degradation to a level that is below *MinConnPath* in many scenarios.

7. RELATED WORK

Many systems bond multiple links or paths into a single higher-performance communication channel. Our work differs primarily in its context and the generality of the problem tackled – we bond multiple paths with disparate delays, capacities, and loss rates. While it is difficult to list all previous works, we note that most existing works assume identical links (e.g., multiple ISDN lines) [10], identical delays [30], or ignore losses [14, 9, 27, 25].

A few systems stripe data between end hosts across arbitrary paths by using TCP or a protocol inspired by it along each path [15, 23]. This provides automatic loss recovery and capacity estimation for each path. These mechanisms work well in an end-to-end setting but not in our in-network proxy setup because loss recovery in them is based on receiver feedback. If applied to our case, such an approach would be futile at hiding losses from users' TCP because of the high delay of paths between the two proxies (§6.2).

MAR [27] and Horde [25] are closest to PluriBus. Both combine multiple wide-area wireless links to improve Internet connectivity on vehicles. MAR uses a simple connection-level striping policy but leaves open the task of building more sophisticated algorithms. We build on their insights to develop a packet-level striping algorithm and show that it significantly outperforms connection-level striping. Horde [25] specifies a QoS API and stripes data as per policy. It requires that applications be rewritten to use the API, while we support existing applications. Neither MAR nor Horde focus on loss recovery.

Delay-based path selection across wireless links was originally proposed in [9]. However, the authors did not build a system around the algorithm, nor did they consider the impact of loss. We show that loss recovery is important for delay-based striping to be effective.

There has also been much work on improving TCP performance over paths involving wireless links, which can be divided into three classes. The first class relies on access to end hosts (e.g., [5]). It cannot be applied to our setting because we do not have access to either vehicular clients or their remote peers. The second class relies on access to wireless provider's infrastructure, such as basestations, for instance, to quickly react to losses [1, 7, 8]. We cannot use this approach either.

The third class uses in-network proxies. The canonical approach here is Split TCP [20] which uses one or more proxies to break end-to-end TCP connections into multiple segments, each running its own TCP connection. This approach works well if lossy segments have a low round trip delay so that lost data can be quickly recovered, without incurring end-to-end delay. In our setting, because we do not have access to wireless carriers' networks, we can only split the TCP connections at the VanProxy and the LanProxy. While this segment tends to be very lossy, it is also responsible for the majority of the end-to-end delay. As a result, Split TCP does not offer any advantage over an end-to-end TCP connection and performs worse than PluriBus (which has faster, erasure coding based loss recovery). We have verified both these behaviors by implementing Split TCP but omit detailed results to due to space constraints.

In the WiFi context, ViFi [3] improves the underlying connectivity from moving vehicles. In contrast, we focus on wide-area wireless technologies and assume that the underlying connectivity is outside of our control. We then develop mechanisms that enable good application performance on top of these links. Systems such as MORE [6] and COPE [19] use coding to reduce packet losses. These systems target the very different setting of multi-hop wireless networks and rely on the ability of WiFi nodes to overhear nearby transmissions.

Evolution codes are inspired by Growth codes [18] that were designed to preserve data in large sensor networks with failing sensors. Many of their design assumptions are specific to their target domain. For example, they assume that the receiver starts out with no information and also assume that multiple senders are attempting to communicate with a single receiver.

8. DISCUSSION

An unconventional aspect of our design is that we aggressively use spare capacity, without worrying about efficiency. In a recent position paper [24], we discussed broadly the value of such an approach and argued that it can be useful only if the overhead of aggressive resource usage can be controlled. PluriBus is a practical instantiation of this approach for the vehicular setting. It minimizes overhead by using opportunistic transmissions for coded packets.

Its aggressive addition of redundancy is based on a selfish perspective. Bus operators typically subscribe to a fixed-price, unlimited usage plans. Our design strives to maximize user performance given that it does not cost more to send more.

In the long-term, however, a natural concern is that PluriBus will lead to higher prices if it increases providers' operational cost. However, there are two reasons why we believe that an exorbitant increase in operational cost would not occur. First, the traffic generated by users on buses may represent a small fraction of the total traffic that the provider carries. Second, even though PluriBus logically fills the pipe, in practice it is not constantly transmitting because it encodes only over data in the last round trip time. As we showed earlier, for realistic workloads PluriBus increases usage by a factor of 2. Finally, we expect that the bus operators would be willing to pay extra for better performance. The cost of wireless access is a small fraction of their operational budget and amortizes over many users.

Another concern is that overly active transmitters may hurt the performance of the other users of the wide-area wireless technology. We believe that the MACs of these technologies and the two reasons mentioned above provide sufficient protection against such concerns.

9. CONCLUSIONS

We designed and deployed PluriBus, a system to provide high-performance Internet connectivity on-board moving vehicles. It seamlessly combines multiple, heterogeneous wide-area wireless paths into a single, reliable communication path. The key novel technique in PluriBus is *opportunistic erasure coding*. Coded packets are sent only when there is instantaneous spare capacity along a path. Packets are coded using Evolution codes that greedily maximize the expected yield of each coded packet by explicitly taking into account what might already be present at the receiver. Our evaluation shows that PluriBus reduces the median flow completion time by a factor of 2.5 for realistic workloads.

10. REFERENCES

[1] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz. Improving TCP/IP performance over wireless networks. In *MobiCom*, Nov. 1995.

[2] M. Balakrishnan, T. Marian, K. Birman, H. Weatherspoon, and E. Vollset. Maelstrom: Transparent error correction for lambda networks. In *NSDI*, Apr. 2008.

[3] A. Balasubramanian, R. Mahajan, A. Venkataramani, B. N. Levine, and J. Zahorjan. Interactive WiFi connectivity for moving vehicles. In *SIGCOMM*, Aug. 2008.

[4] P. Barford and M. Crovella. Generating representative Web workloads for network and server performance evaluation. In *SIGMETRICS*, 1998.

[5] S. Biaz and N. Vaidya. Discriminating congestion losses from wireless losses using inter-arrival times at the receiver. Technical Report 98-014, Texas A&M University, 1998.

[6] S. Chachulski, M. Jennings, S. Katti, and D. Katabi. Trading Structure for Randomness in Wireless Opportunistic Routing. In *SIGCOMM*, 2007.

[7] R. Chakravorty, S. Katti, J. Crowcroft, and I. Pratt. Using TCP flow aggregation to enhance data experience of cellular wireless users. *IEEE JSAC*, 2005.

[8] M. C. Chan and R. Ramjee. TCP/IP performance over 3G wireless links with rate and delay variation. *Wireless Networks (Kluwer)*, 11(1-2), 2005.

[9] K. Chebrolu and R. Rao. Bandwidth aggregation for real-time applications in heterogeneous wireless networks. *IEEE TOMC*, 4(5), 2006.

[10] J. Duncanson. Inverse multiplexing. *IEEE Communications Magazine*, 32(4), 1994.

[11] J. Eriksson, S. Agarwal, P. Bahl, and J. Padhye. Feasibility study of mesh networks for all-wireless offices. In *MobiSys*, 2006.

[12] Cira 3g mobile broadband router. http://www.feeneywireless.com/products/routers/cira/cira.php.

[13] M. Han, Y. Lee, S. Moon, K. Jang, and D. Lee. Evaluation of VoIP quality over WiBro. In *PAM*, April 2008.

[14] A. Hari, G. Varghese, and G. Parulkar. An architecture for packet-striping protocols. *ACM TOCS*, 17(4), 1999.

[15] H.-Y. Hsieh and R. Sivakumar. A transport later approach for achieving aggregate bandwidths on multi-homed mobile hosts. In *MobiCom*, Sept. 2002.

[16] N. Hu, L. E. Li, Z. M. Mao, P. Steenkiste, and J. Wang. Locating Internet bottlenecks: Algorithms, measurements, and implications. In *SIGCOMM*, Aug. 2004.

[17] M. Jain and C. Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. In *SIGCOMM*, Aug. 2002.

[18] A. Kamra, V. Misra, J. Feldman, and D. Rubenstein. Growth Codes: Maximizing Sensor Network Data Persistence. In *SIGCOMM*, 2006.

[19] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Medard, and J. Crowcroft. XORs In The Air: Practical Wireless Network Coding. In *SIGCOMM*, 2006.

[20] S. Krishnamurthy, M. Faoutsos, and S. Tripathi. Split TCP for Mobile AdHoc Networks. In *GlobeComm*, 2002.

[21] Y. Liao, Z. Zhang, B. Ryu, and L. Gao. Cooperative robust forwarding scheme in DTNs using erasure coding. In *MILCOM 2007*, Oct. 2007.

[22] M. Luby. LT Codes. In *FOCS*, March 2002.

[23] L. Magalhaes and R. Kravets. Transport level mechanisms for bandwidth aggregation on mobile hosts. In *ICNP*, Nov. 2001.

[24] R. Mahajan, J. Padhye, R. Raghavendra, and B. Zill. Eat all you can in an all-you-can-eat buffet: A case for aggressive resource usage. In *HotNets*, 2008.

[25] A. Qureshi and J. Guttag. Horde: separating network striping policy from mechanism. In *MobiSys*, June 2005.

[26] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *SIAM J. Appl. Math.*, June 1960.

[27] P. Rodriguez, R. Chakravorty, J. Chesterfield, I. Pratt, and S. Banerjee. MAR: A commuter router infrastructure for the mobile Internet. In *MobiSys*, June 2004.

[28] S. Sanghavi. Intermediate performance of rateless codes. In *Information Theory Workshop*, 2007.

[29] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The end-to-end effects of Internet path selection. In *SIGCOMM*, Aug. 1999.

[30] A. Snoeren. Adaptive inverse multiplexing for wide-area wireless networks. In *Globecom*, Dec. 1999.

[31] N. Spring, R. Mahajan, and T. Anderson. Quantifying the causes of path inflation. In *SIGCOMM*, Aug. 2003.

[32] More cities offer Wi-Fi on buses. http://www.usatoday.com/tech/wireless/2008-04-10-wifi_N.htm.

[33] Amtrak and T-Mobile offer WiFi at select stations, sounder trains have free WiFi. http://blogs.zdnet.com/mobile-gadgeteer/?p=642.

[34] WiFi incompatibility on Seattle metro? http://www.internettablettalk.com/forums/archive/index.php?t-18539.html.

[35] The last fifty feet are WiFi. Reader comments at http://www.bwianews.com/2007/04/the_last_fifty_.html.

[36] Sound transit riders. A mailing list for Microsoft employees.

[37] Google's buses help its workers beat the rush. http://www.nytimes.com/2007/03/10/technology/10google.html.

[38] Metro bus riders test county's first rolling WiFi hotspot. http://www.metrokc.gov/kcdot/news/2005/nr050907_wifi.htm.

[39] Microsoft giving workers free ride – with its own bus service: WiFi-enabled system will debut this month. http://seattlepi.nwsource.com/business/330745_msfttranspo07.html.

[40] Wi-Fi bus connecting the streets of San Francisco. http://www.switched.com/2008/02/22/wi-fi-bus-connecting-the-streets-of-san-francisco/.

[41] Free Wi-Fi on Sound Transit. http://www.soundtransit.org/x6083.xml.