E PluriBus Unum: High Performance Connectivity On Buses

Ratul Mahajan Jitendra Padhye Sharad Agarwal Brian Zill
Microsoft Research

Abstract – We present PluriBus, a system that enables high-performance Internet access on-board moving vehicles. It seamlessly bonds multiple wide-area wireless links, which individually tend to be lossy and have high delays, into a reliable communication channel. PluriBus employs a novel technique called *opportunistic erasure coding*. It sends erasure coded packets only when there is an opening in a path's spare capacity. Packets are coded using *Evolution* codes that we have developed to greedily maximize the expected number of packets recovered with each coded packet. These methods let us opportunistically use any spare resources, in a way that minimizes the impact on data traffic. We have deployed PluriBus on two buses. Our experiments show that 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 paths.

1. INTRODUCTION

Internet access on-board buses, trains, and ferries is becoming increasingly common [41, 46, 48, 49]. Public transportation agencies in over twenty cities in the USA currently provide such access to boost ridership; many more are planning for it [40]. Corporations also provide such access on the commute vehicles for their employees [45, 47]. For instance, more than one-quarter of Google's work force in the Bay Area uses such connected buses [45]. 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 their increasing popularity and their unique environment, insufficient attention has been paid in the research community 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 [42, 43, 44]. In fact, 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 shows the typical way to enable Internet access on buses today. Riders use WiFi to connect to a device [13, 20, 25, 39] on the bus, which we call *VanProxy*. The device provides Internet access using a wide-area wireless technology such as EVDO or HSDPA. In some cases, to support a large number of users, multiple

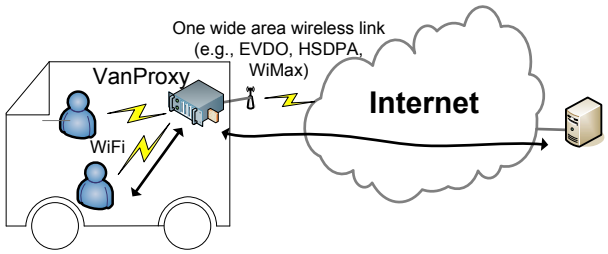


Figure 1: Existing architecture for providing connectivity on-board buses.

VanProxy devices are employed. These devices operate independently; users connect to one of them, typically based on WiFi signal strength. The WiFi part of existing setups tends to be reliable because of its small coverage region. But the quality of the wide-area wireless connectivity is critical for good performance.

Our measurements across multiple technologies confirm earlier findings [32, 14] that wide-area wireless links offer poor service from moving vehicles. They have high delays, lack capacity (especially in the uplink direction), and frequently drop packets. Occasionally, they even suffer blackouts, i.e., periods with very high loss rate. Poor application performance in this environment 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 2, it uses multiple wide-area wireless links. It bonds them with the help of a machine connected to the wired network, which we call *LanProxy*.

PluriBus employs two techniques to boost application performance. First, it uses a novel technique called *opportunistic erasure coding* to mask losses from applications. This technique differs from existing erasure coding methods in when and how many erasure coded packets are sent as well as what each coded packet carries.

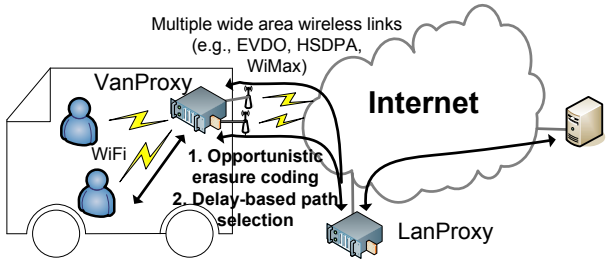


Figure 2: The architecture of PluriBus.

Coded packets are sent only during instantaneous openings in bottleneck link's available capacity. The openings are judged using an estimate of queue length and capacity. Done this way, coded packets do not delay or steal bandwidth from data packets and provide as much protection as available capacity allows. In constrast, existing methods add a fraction of coded packets per data packet that is independent of load and available capacity [3, 26]. These may not provide sufficient protection even when additional capacity is available or may slow down data traffic when the load is high [29].

In PluriBus, packets are coded using *Evolution codes* that we have designed to greedily maximize the expected number of data packets recovered with each coded packet. Evolution codes thus stress partial recovery of as many data packets as possible. In contrast, existing codes, such as Reed-Solomon [31] or LT codes [27], stress full recovery. That is, they aim to minimize the number of coded packets needed at the receiver to recover all data packets. Given the burstiness of incoming traffic and losses, it is hard to guarantee at short time scales that the required number of coded packets will be received. And when that does not happen, very little data may be recovered by these codes [33].

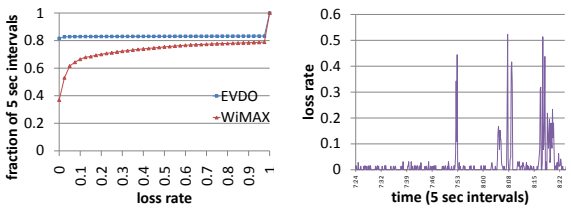
Our second technique is to stripe data across the paths offered by the various links based on an estimated delivery delay along each path [10]. We estimate delivery delay by estimating path capacity, queue length, and propagation delay. The striping decision for each packet is taken independently. Done this way, PluriBus does not use a slower path until the queues on a faster path increase its delay to match the slower path.

PluriBus has been deployed on two of Microsoft's campus buses for two months. Each bus is equipped with two wide-area wireless links, one of which is EVDO and the other is WiMax. Microsoft's IT department is currently evaluating PluriBus for operational use.

We evaluate PluriBus using our deployment as well as controlled experiments with an emulator. 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 multiplexing traffic across multiple links. We also study the two techniques individually. Opportunistic erasure coding by itself improves performance by a factor of 1.6. Interestingly, delay-based path selection by itself hurts performance; the preferred lower-delay link happens to have a higher loss rate in our deployment. After losses have been countered using opportunistic erasure coding, delay-based path selection improves performance by a factor of 1.6.

2. THE VEHICULAR ENVIRONMENT

In this section, we briefly characterize paths and workload of the vehicular environment that we target. 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 and the traffic is highly bursty. These characteristics motivate our solution, which is presented in the following sections.



(a) (b)

Figure 3: (a) The CDF of loss rate from the wired network to the buses for paths over the two links. (b) A chosen hour-long window that shows the temporal behavior of loss rate for WiMax.

2.1 Our Testbed

To study path characteristics from a moving vehicle, we equipped two buses that ply around the Microsoft campus with desktop computers. The buses operate approximately from 7 AM to 7 PM on weekdays. The onboard 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 the path quality by sending packets through each provider's network between the bus and a computer deployed inside the Microsoft corporate network. 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 3(a) shows the CDF of loss rates, averaged over 5 seconds, from the Microsoft 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. These intervals are referred to as blackouts. Figure 3(b) shows one hand-picked hour-long window with a particularly bad loss behavior. These results agree with earlier measurements from moving vehicles [32, 14].

Such lossy behavior can hurt many applications, especially those that use TCP. Fortunately, we also find that high-loss periods of the two providers are roughly inde-

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

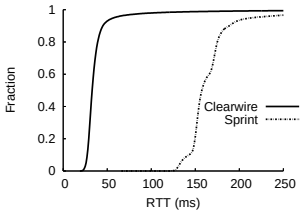
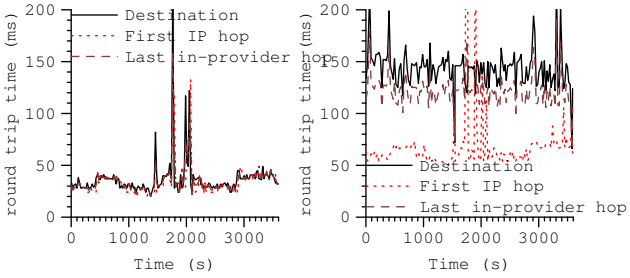


Figure 4: The CDF of RTT for paths over the two links.



(a) WiMax (b) EVDO

Figure 5: Breakdown of RTT for paths over the two links.

pendent [32], which offers hope that performance can be improved by bonding links from different providers.

2.2.2 Paths have high and disparate delays

Figure 4 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 time is spent, we run traceroute from the bus to the wide-area host. We extract from this data the RTTs to the destination, to the first IP hop, and to the last hop in the wireless provider's network. The exit point is inferred using DNS names of the routers [37].

Figure 5 shows the results for an hour-long window. Surprisingly, a third of the delay for EVDO and nearly all of the delay for WiMax is to the first IP hop. For both, nearly all of the delay is inside the provider network. As we discuss later, this observation has implications regarding 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. The delay disparity between paths may be countered by bonding links from the same provider, but that reduces system reliability [32].

2.3 Workload Characteristics

Duration (hours)	3.34
Packets	354,465
Bytes	286,266,708
UDP flows	833
TCP flows	3,796
Avg. number of devices	4.09

Table 1: Aggregate characteristics of 11 traces

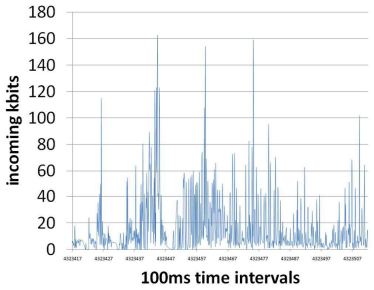


Figure 6: Traffic arriving for the clients from the Internet during 100 seconds

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, 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 basic statistics of these traces are summarized in Table 1.

We find the essential characteristics of this workload to be similar to those in many other environments. Traffic is dominated by short TCP flows, which are especially vulnerable to packet loss. The traffic is also highly bursty, as illustrated by the example 100-second period shown in Figure 6. Burstiness makes it hard to accurately predict short-term traffic intensity or leftover capacity. This factor makes it hard to use existing erasure coding methods because it is hard to estimate how much redundancy can be added without overloading the path.

3. APPROACH

Given the poor quality of wide-area wireless connectivity from moving vehicles, how can we best improve application performance? We could urge the carriers to improve the quality of the underlying connectivity. Doing so will probably require significant investment on their part, to improve coverage and perhaps deploy better handoff protocols. Furthermore, this is a longer-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.

3.1 Architecture

The PluriBus architecture is shown in Figure 2. In contrast to the existing architecture (Figure 1), we equip VanProxy with multiple wireless links. Using multiple links offers additional capacity when needed and boosts reliability [32], for instance, when one of the links is experiencing a blackout.

Additionally, we relay all packets through *LanProxy*, a server located on wired Internet. Relaying packets through *LanProxy* allows us to mask packet losses, which is not possible if the VanProxy sends packets directly into the Internet. It also allows us to flexibly stripe data across the multiple links. Without the LanProxy, striping must be done at the level of individual connections because the different links on the VanProxy are assigned different IP addresses by the providers. Connection level striping is known to be suboptimal in terms of load distribution [36]. It is also less reliable in face of blackouts. In §6.2, we compare packet-level striping of PluriBus with a connection-level striping policy.

Relaying through LanProxy may increase the end-to-end latency of the traffic. Interestingly, we find a significant *Detour* effect [34] for our deployment – routing through the LanProxy reduces end-to-end path latency to a majority of the destinations, perhaps because our LanProxy has better connectivity to the Internet than our wireless providers. While we do not claim that the same will hold for all deployments, the impact of routing through the LanProxy is likely to be relatively small because: i) the delay inside the wireless provider's network, which is common to both direct and indirect (through LanProxy) paths, tends to be high (§2.2.2); ii) the LanProxy is deployed in the same city as the provider's network, and Internet path latencies within the same city tend to be small [37].

3.2 Solution Overview

The problem we address can be concisely stated as follows. We are given one or more paths between the VanProxy and LanProxy. Each path is based on a different wide-area wireless link, is lossy with a time-varying loss rate, and has an almost fixed or slowly-varying transmission capacity. Different paths have different capacities and delays. The incoming data is bursty and arrives at an unknown and time-varying rate.

Our goal is to use these paths to deliver data that arrives at one proxy from either the vehicular clients or Internet hosts to the other proxy. We want to do so in a manner that maximizes performance for interactive traffic such as Web transactions. Thus, we strive to minimize the loss rate and delay experienced by the data packets.

The concept of bonding multiple communication channels into a single channel has been explored in many contexts. As we argue in §7, existing solutions cannot be used for our setting because of the complex nature of the paths. These solutions are based on assumptions, such as that the paths have identical delays, that do not hold in our environment.

Our solution has two components:

1. Opportunistic Erasure Coding to mask losses: There are two possible methods that we can use to hide path losses from end users: i) retransmit lost packets based on feedback from the other proxy; and ii) proactively send redundant packets using erasure coding. As we show in §6.4, retransmission-based loss recovery performs poorly in our environment. This is because paths between the two proxies have a high RTT, which makes loss recovery slow. Additionally, the inter-proxy RTT can be major component of the end-to-end RTT. Because TCP can often detect loss within one end-to-end RTT and retransmissions would take roughly 1.5 times the inter-proxy RTT to reach the other proxy, retransmissions cannot hide many losses from TCP. A race could be created between our and TCP's retransmissions. Erasure coding is thus a better fit for our environment.

We desire two properties from the erasure coding method. The first property dictates how many coded packets are sent and when, and the second one dictates what code is used to generate coded packets. The first property is that coded packets interfere minimally with data packets while providing as much available capacity allows. Coded packets interfere if data packets have to wait behind them in the bottleneck queue because this amounts to them stealing valuable capacity from the data packets. To our knowledge, none of the existing methods, such as Maelstrom [3] or CORE [26], have this 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 the excess capacity at short time scales is difficult.

To fulfill the first property, PluriBus sends 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. Done this way, coded packets always defer to data packets and delay them by at most one packet. In §6.4.3, we show that this method brings negligible slowdown for data traffic. At the same time, our method provides as much protection as the amount of spare capacity allows. Our strict prioritization of data packets over coded packets increases total goodput because while data packets deliver one new packet at the receiver coded packets deliver less than one on average.

Not having a fixed overhead means that the level of

protection drops as the rate of new data increases. This effect is essentially similar to losses that applications suffer today when they send faster than what the path can support. As it happens today, higher-layer congestion control mechanisms will adapt and reduce their rate in response to losses that are exposed to them.

Logically, our opportunistic erasure coding uses all spare capacity, to maximize the degree of protection. We discuss the implications of this behavior in §8. But we note that in practice the path is always not fully used because coded packets are sent only when new data packets arrived within the last RTT.

The second property is that the code not rely on a threshold number of coded packets be generated at the source or received by the receiver. Given the burstiness of incoming traffic and thus available capacity, such guarantees are hard to provide in our setting. Conventional erasure codes, whether rateless (e.g., LT [27]) or otherwise (e.g., Reed-Solomon [31]), do not have this property. These codes encode k data packets into more than k coded packets. As long as the receiver receives a threshold number of the coded packets, it can recover all k packets. However, when fewer than this threshold are received, very little is recovered [33].

Instead of aiming for a full recovery, as existing codes do, we design a new code, called *Evolution* code, that stresses *partial recovery*. In our code, each coded packet greedily maximizes the expected number of data packets that will be recovered. We show in §6.4 that Evolution codes perform better than codes optimized for full recovery. Conversely, greedy partial recovery makes Evolution codes less efficient for full recovery. They need more coded packets to be received to recover all data packets. Simulations, not presented in this paper, reveal that the loss in efficiency is only 15-20%.

Additionally, we encode over a moving window of packets that arrived at the sending proxy in roughly the last RTT. (Recovering older packets is wasteful.) Evolution codes naturally apply to moving windows. It is not clear how to adapt many of the existing erasure codes to moving window of packets.

2. Delay-based path selection PluriBus sends each data packet along the path that is likely to deliver it first [10, 30]. This method naturally generalizes striping mechanisms such as round robin to the case of paths with different delays and capacities. It continues to send traffic along the shortest path until the queue length on this path brings its delay up to the level of the next shortest path, and so on. This method also makes it less likely for a later packet to arrive before a previously sent packet. Due to variation in link delays, some reordering may still happen. We use a small re-sequencing buffer to address this possibility.

While the two components of our solution combine well to offer high performance connectivity for moving vehicles, they can be independently used as well. Opportunistic erasure coding can boost reliability even when there is only one wide-area wireless link. Similarly, delay-based path selection can bond multiple paths such that the delay of the combined channel is low even when erasure coding is not used to boost reliability.

4. DESIGN OF PluriBus

We now describe our design in more detail. Without loss of generality, we consider traffic headed in one direction only. Identical algorithms run in each direction. For the purpose of this section, the terms sender and receiver refer to the two proxies shown in Figure 2.

PluriBus transmits data packets as soon as they arrive at the sending proxy, along the path that is 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.

4.1 Evolution code

Evolution code aim for greedy partial recovery, by maximizing the expected number of packets that will be recovered with each transmitted coded packet. At any given instant, the sender codes over a set of data packets W that were sent within the previous round trip time. Assume that the sender also estimates the fraction r of the W packets (but not which exact packets) that has been successfully recovered by the receiver, based on past transmissions of data and coded packets. 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 based on the contents of earlier packets and the paths over which they were sent. We describe later how W and the fraction r are updated at the sender.

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 in W. The probability that this coded packet will yield a previously missing data packet at the receiver is equal to the probability that exactly one out of the x packets is lost. That is, the expected yield $Y(x)$ of this packet is:

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

$Y(x)$ is maximized for $x = \frac{-1}{ln(r)}$.

The result of this analysis can be intuitively explained. If the expected number of data packets at the receiver (r) 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. essentially re-send one of the packets in W); coding even two is likely futile as the chance of both being absent, and hence of nothing being recovered, is high. Conversely, if more packets are already there at the receiver, encoding a higher number of packets is the most 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 (§6.4.2). Furthermore, if $|W| > 1$ and $\lfloor \frac{-1}{ln(r)} \rfloor \geq |W|$, we XOR only $|W| - 1$ packets. We never XOR the entire window of packets because of a subtle corner case that arises if 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, the direct application of the analysis would lead to the repeated transmission of the same deterministic coded packet at each opportunity. But this particular coded packet cannot recover anything new.

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

$i)$ When a new data packet is sent, it is first added to W, and then:

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

where 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. The estimate of the current loss rate at the sender may not be accurate. But, as we show in § 6.4.2, the performance of evolution codes is robust this inaccuracy.

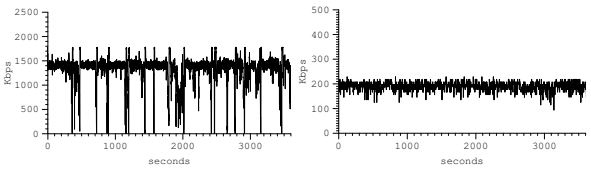
$ii)$ When a coded packet, formed by XOR'ing x data packets, is sent, W does not change, and

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

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, and r is unchanged. This step ensures that the sender encodes only over roughly one round trip of data.

At this point, we can explain the rationale for the name "Evolution." 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. It increases as more coded packets are generated (and r increases) and decreases when new data is included in the window. In this respect, Evolution codes are a generalization of Growth code [21].



(a) Downlink (b) Uplink

Figure 7: The downlink and uplink throughput of WiMax paths. The y-axis range of the two graphs is different.

4.2 Estimating Queue Length

We maintain queue-length along a path in terms of time required for the bottleneck queue to fully drain. It is zero initially. It is updated when the system polls for the current estimate:

$$Q_{new} = max(0, Q_{old} - TimeSinceLastUpdate)$$

It is also updated after sending a packet:

$$Q_{new} = max(0, Q_{old} - TimeSinceLastUpdate) + \frac{PacketSize}{PathCapacity}$$

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.

Estimating capacity for wide-area wireless paths is a simpler proposition than doing so for WiFi. The MACs (media access control) of these links reserve a set amount of media capacity per client by tightly controlling all packet transmissions at the base station. Unlike the WiFi MAC, this behavior also isolates clients from each other, and clients do not usually interfere with each others' transmissions. As an example, Figure 7 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 the dips in throughput are due to packet losses and not slowdowns in queue drain rate.

While the behavior above suggests that we could simply configure path capacities, we include an estimation component to be robust to any variations. We use the insight behind recent bandwidth measurement tools [17, 18]: 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 create packet trains with a rate faster than path capacity.

We bootstrap nodes with expected capacity of the paths. The receiver then watches for changes in path capacity. 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 esimate, we show in §6.5 that our path delay estimate, which is based in part on this estimate, is fairly accurate.

4.3 Identifying Minimum Delay Path

To send newly arrived data packets, PluriBus needs to estimate the current delay along each 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, as we show in §6.5, 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. 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 capacity of bottleneck link; ii) time spent in the queue; and iii) propagation delay. We described above how we estimate the first two quantities. 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 do not need absolute delay values; we can identify the faster path even if 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 paths be d_1 and the (unknown) skew between the two clocks be δ. A packet that is sent by the sender along the path at local time s_1 will be received by the receiver at local time

$$r_1 = s_1 + \delta + d_1 + QueueLength_1 + \frac{PacketSize}{PathCapacity_1}$$

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

$$d_1 + \delta = r_1 - s_1 - \frac{PacketSize}{PathCapacity_1}$$

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 $r_i - s_i - \frac{PacketSize}{PathCapacity_i}$ for each path, which corresponds to $(d_i + \delta)$. 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 $\frac{PacketSize}{PathCapacity_i}$ 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. Observe that we ignore clock drift, which is safe as long as it is not too high to change the skew at small timescales, such that packets sent along different paths witness different skews. Typical clocks today easily meet this criterion.

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 $\frac{PacketSize}{PathCapacity_i} + QueueLength_i + (d_i + \delta)$. This sum is in fact an estimate of the local time at the receiver when the packet will be delivered. We show in §6.5 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 initializes its wide-area wireless interfaces and uses one of them to contact the LanProxy at the configured IP address. It informs the LanProxy of the current IP addresses that it obtained from the wide-area wireless providers. The LanProxy sends configuration information to the VanProxy, including the IP address range that the VanProxy should use for clients in the vehicle and addresses for DNS servers. Clients use DHCP to get their configuration information from the VanProxy.

The two proxies essentially create a bridge between themselves by tunneling packets over the paths that connect them. When contacting a remote computer, a client sends its IP packet to the VanProxy which encapsulates the IP packet into a UDP packet, and also adds a custom header (described below). The IP-in-UDP encapsulation is needed because both of our wide area wireless providers simply discard IP-in-IP encapsulated packets entering their network. In fact, the packets sent by the LanProxy have to masquerade as DNS responses to get

past their firewalls. The VanProxy then sends the encapsulated packet to the LanProxy using the wide-area interface with the least estimated delay. The LanProxy decapsulates the packet to recover the client's original IP packet and relays it to the remote computer.

In the reverse direction, packets from the remote computer to a vehicular client reach the LanProxy. Since the LanProxy may be serving multiple VanProxies, the packet's destination IP address is used to identify the intended VanProxy. The LanProxy encapsulates and sends the packet to the target VanProxy on one of its wide-area interfaces based on the delay estimates. The VanProxy decapsulates the packet and sends it to the target client. In this architecture, either proxy can also perform network address translation (NAT) for the clients if needed.

PluriBus uses multiple sequence number spaces at each proxy. One space is used for data packets that arrive at the proxy to be sent to the other proxy; each data packet is assigned the next sequence number from this space. These data-level sequence numbers let the receiver uniquely identify data packets and their relative order. In addition, there is also a per-path sequence number space; each packet transmitted along a path is assigned the next sequence number from this space. Path-level sequence numbers help the receiver estimate various properties of the incoming path, such as loss rate. The LanProxy maintains this set of spaces for each VanProxy that it serves.

Each PluriBus proxy caches all data packets arriving from other proxies for a brief window of time so that coded packets can be decoded. They also have a sequencing buffer to order received data packets. When a data packet received from the other proxy has a sequence number that is higher than one plus the highest sequence number relayed, it is stored in this buffer. It leaves the buffer as soon as the missing data packets are received directly or recovered from coded packets. If the hole is not filled for a threshold amount of time, set to 50 ms in our experiments, the data packet is relayed immediately.

The header fields of PluriBus are shown in Table 2. There are three types of messages: i) pure data packets; ii) coded packets; and iii) control packets. The data-level sequence number in a coded packet corresponds to the highest sequence number encoded in that packet. Coded packets contain an additional 4-byte bitmap that encodes which other packets are contained in it, relative to the highest sequence number. Control packets are exchanged between the proxies to exchange configuration information, report on wide-area address changes at the the VanProxy, and properties of incoming paths.

These extra header fields, and the IP-in-UDP encapsulation lowers the effective link MTU by 41 bytes. If the senders do not learn of the lower MTU, some of the packets they send may have to be fragmented at the proxies. We inform clients on the bus of the lower MTU via the *Interface MTU option* [1] in the DHCP response. Informing external senders is harder as Path MTU discovery is not always used on the wide area Internet. Fortunately, clients with a lowered interface MTU will inform their wide area peers of this fact during TCP connection establishment via the MSS option of a TCP SYN packet. For other clients, we are experimenting with modifying the MSS option of TCP SYNs as they pass through the VanProxy. Obviously, this approach only applies to TCP traffic, but suffices for over 99% of our traffic.

Message type	1 byte
Timestamp (integer milliseconds)	2 bytes
Data-level sequence number	2 bytes
Path-level sequence number	2 bytes
Last data-level sequence number received	2 bytes

Table 2: The header fields used for communication between the proxies.

6. EVALUATION

In this section, we evaluate PluriBus. Along with studying the performance of the overall system in §6.2, we study in detail the behaviour of opportunistic erasure coding in §6.4 and of delay-based path selection in §6.5.

6.1 Methodology

We employ a mix of experiments using our deployment (§2.1) and an emulator. The two platforms provide complementary benefits. The deployment lets us evaluate the benefit of PluriBus in real environments. Emulation lets us control the environment and isolate individual factors. It uses the same implementation as the deployment.

Workload For the experiments presented in this paper, we generate realistic, synthetic workloads from the traces described in Section 2.3, using a methodology adapted from [12]. We first process the traces to obtain distributions of flow sizes and inter-arrival times, where a flow is identified using the combination of the two IP addresses, the two ports, and the protocol field. The synthetic workload is based on these distributions of flow sizes and inter-arrival times.

To study the performance of PluriBus as a function of load, we also synthesise 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 flow 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, we find that the performance

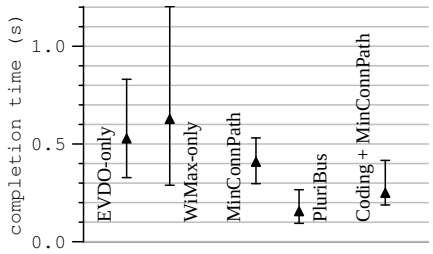


Figure 8: Performance of various systems for a workload based on our traces. [Deployment]

of a synthetic workload scaled by a factor of 1 is similar to an exact replay of flow size and arrival times.

To verify that our conclusions apply broadly, we have also experimented with 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. We obtain qualitatively consistent results with these workloads but omit details due to space constraints.

A connection-level striping policy for comparison To place the performance of PluriBus in context, we also consider a connection-level striping policy that can be considered as the state-of-art for data striping in vehicular settings [32]. In this policy, a new connection is mapped to the path with the minimum number of active connections. No special loss recovery is performed. We refer to this policy as *MinConnPath* and use it as a representative of the class of policies that stripe at the level of connections, i.e., they do not send the packets of a connection over different links.

MinConnPath performs better than other connection-level policies that we experimented with. Round robin does worse than *MinConnPath* because it is does not consider current load on the link while mapping new connections and in some situations leads to a highly unbalanced distribution. Another policy that we tested maps incoming connections to links that currently carry less load. This policy too performs worse than *MinConnPath*. 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.

Performance measure The primary measure of performance that we use is flow completion time. This is of direct interest to interactive traffic such as short Web transfers that dominates in our traces (§2.3). We use the median as the representative measure. In many cases, we also show the inter-quartile range (25-75%) of completion times, which captures the observed spread.

6.2 Overall performance

Figure 8 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 (25%-75%) of the connection completion times. These results are based on our deployment on-board buses, and each configuration ran for at least two days during which time they complete tens of thousands of connections.

The two bars on the left show the performance of the system when only one of the wireless uplinks is used directly, as is the norm today. We see that the median completion time of EVDO is 500 ms and of WiMax is 700 ms. WiMax offers lower performance even though it has a much lower round trip time because of its higher loss rate. The observed overall 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 suffer a loss.

The third bar shows the performance when both links are employed simultaneously by *MinConnPath*, which, as we have explained earlier, represents the state-of-the-art. We see that *MinConnPath* improves performance over using only one link, bringing the median completion time to 400 ms, by spreading the load across both links. It reduces the completion time spread as well because fewer connections suffer heavy losses on the WiMax link. Because it balances at the connection level, as long as more connections are active on the WiMax link, it uses the EVDO link for new connections.

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 the 400 ms observed for *MinConnPath*.

To understand the sources of this performance improvement, let us consider how PluriBus differs from *MinConnPath*. There are two primary differences between PluriBus and *MinConnPath*. First, PluriBus uses opportunistic erasure coding to reduce the impact of packet losses. Second, PluriBus uses a per-packet delay-based striping policy. Both these mechanisms contribute to the better performance of PluriBus.

To tease apart the contribution of the two mechanisms, we added opportunistic erasure coding to the *MinConnPath* policy. The performance of this policy is shown by the fifth bar in Figure 8. We see that the by adding coding, the median completion time is reduced to 250 ms, which represents a reduction of 150 ms over the median for *MinConnPath*, or an improvement factor of 1.6.

Note however, that the median completion time of PluriBus, is 150 ms, which represents a reduction of 100 ms over the median for *CodedMinConnPath*. It follows that this improvement can be attributed to the per-packet, delay-based striping policy used by PluriBus. since it is the only difference between *CodedMinConnPath* and PluriBus.

In summary, we have shown that PluriBus performs sig-

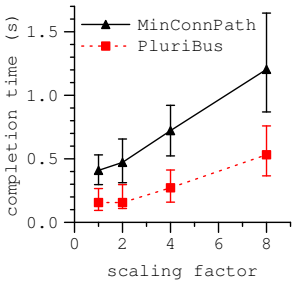


Figure 9: Performance of *MinConnPath* and PluriBus as a function of load. [Deployment]

nificantly better than the state-of-the-art *MinConnPath* system. Furthermore, we have also characterized the contribution o of opportunistic erasure coding and per-packet delay-based striping to this improvement.

However, several questions remain to be answered. For example: (i) How well does PluriBus perform under increased load, given that there are fewer opportunities to send coded packets at higher load? (ii) Could we have used some other coding scheme besides evolution codes? (iii) What is the overhead of our aggressive coding policy? (iv) How does *MinConnPath* alone compare against delay-based striping in absence of coding? In the rest of the section, we answer these and other related questions.

6.3 Impact of load

We now study the performance of PluriBus at higher loads, when PluriBus gets fewer opportunities to send erasure coded packets. Figure 9 shows the median and interquartile 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 we scale the workload by a factor of eight. At the extreme, PluriBus reduces the median completion time by a factor of 2.

6.4 Opportunistic erasure coding

In this section, we study the behaviour of opportunistic erasure coding in detail. We compare it to other potential methods for masking packet loss, evaluate the inaccuracy of loss rate estimation in vehicular environments and its impact on performance, and quantify the impact of aggressive coding.

We use workloads and performance measures similar to those in the last section. In a separate set of experiments, omitted from this paper, we have evaluated using detailed packet-level simulations the behaviour of Evolution codes and how they compare with LT codes and Reed-Solomon codes in terms of recovering lost packets.

6.4.1 Benefit relative to other methods

The last section shows that the loss protection method of PluriBus significantly improves performance. We now study how it compares to other potential methods for loss protection. One of these methods retransmits lost packets based on receiver feedback. Comparison with it shows the value of using erasure coding for loss protection.

We also study 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 pure packets is fixed, independent of prevailing workload and available capacity. Comparison with it shows the value of using opportunistic transmissions to guard against losses rather than using a preset overhead. The second method is based on rateless codes such as LT that can be adapted for opportunistic usage because of their ability to generate on-demand as many coded packets as needed. The complexity (i.e., the degree) of coded packets generated by these codes is independent of loss rate and what might be already present at the receiver. Comparison with it shows the value of using Evolution code to guide the complexity of the coded packet.

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; when $K = 10$, every 11^{th} packet is coded and codes over the previous 10 packets that still remain in the unacknowledged window. This method is a simple version of fixed-overhead codes. It is equivalent to $(K, 1)$ Maelstrom code [3].

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 the same as that used in LT codes. 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 our experiments.

For a controlled environment, we conduct this experiment using a network emulator. We configure one link between VanProxy and LanProxy. 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 find that results with more sophisticated loss models such as Gilbert-Elliot are qualitatively consistent; we omit them due to space constraints.

Figure 10 shows the results as a function of the configured loss rate. For each alternative method, it plots the median completion time divided by the median completion of PluriBus. Values greater than one imply that the

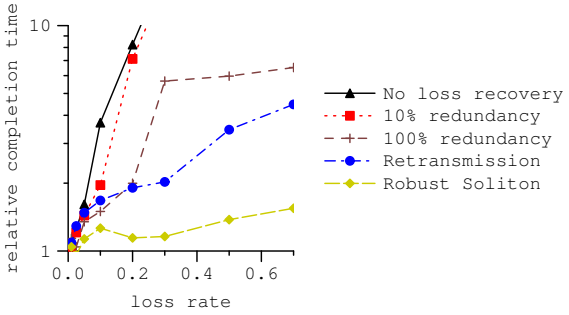


Figure 10: Performance of different loss protection methods relative to the coding method of PluriBus. [Emulation]

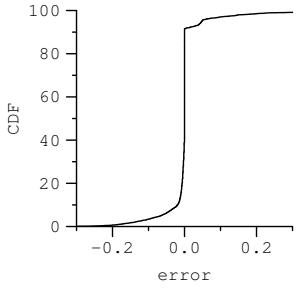


Figure 11: Accuracy of loss rate estimates in PluriBus. [Deployment]

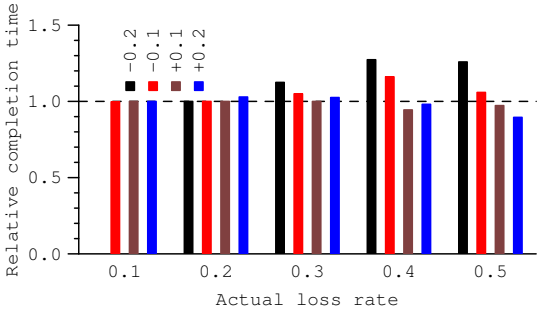


Figure 12: Impact of inaccuracy in loss rate estimate in PluriBus. [Emulation]

other method performs worse. K% redundancy curves correspond to fixed-overhead schemes.

Two broad conclusions can be drawn from this graph. First, unsurprisingly, some form of loss protection can significantly boost performance. For instance, at 10% loss rate the relative completion time without any loss protection is at least twice that for any other form of loss protection.

Second, PluriBus provides the best performance amongst all these methods. (All relative completion time values are greater than one.) It does better than retransmissions because erasure coding is able to recover from losses faster, which improves performance and also makes it less likely to get into race conditions with TCP.

Across all loss rates, PluriBus outperforms both fixed-overhead methods. This is true even at 10% loss rate, where 100% redundancy is able to recover from most losses. The advantage of PluriBus stems from its ability to send coded packets in a way that does not get in the way of pure data packets. Fixed-overhead methods either add too little redundancy at high loss rates or add too much redundancy at low loss rates in a way that slows down pure data packets.

Finally, PluriBus provides a noticeable advantage over using rateless codes. Given that when coded packets are sent is the same for both cases, 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. Because conventional rateless codes are oblivious to this, they are more likely to send a overly complex coded packet that cannot be decoded at the receiver. They are also more likely send very simple packets, which means that they need to send more coded packets to recover lost data.

6.4.2 Accuracy of loss rate estimate

PluriBus estimates the current loss rate on each wireless uplink and uses that estimate to implement Evolution codes. Given the dynamics of the vehicular environment, loss rate maybe hard to estimate accurately. Figure 11 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%.

To understand performance degradation in environments where loss rate may be hard to estimate, we study the impact of using an inaccurate estimate of loss rate in PluriBus. In the same emulator setup as before, which has one link between VanProxy and LanProxy, we program the proxies to use an inaccurate loss rate instead of estimating it based on transmitted packets.

Figure 12 shows the results, when we program the proxies to use loss rate offsets of $\pm$0.1 and $\pm$0.2 off of the actual loss rate on the emulated link. Positive offsets correspond to overestimation of the loss rate and negative offsets correspond to underestimation. In the graph, the x-axis corresponds to the actual loss rate and individual bars correspond to different offset values. There is no bar for the 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 actual loss rates, performance degrades when loss rate is underestimated. This degradation is about 20% when the underestimation is -0.2.

Interestingly, at high loss rates, performance improves if loss rates are overestimated. This stems from our strat-

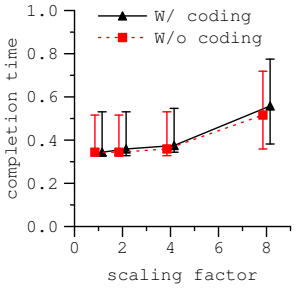


Figure 13: Impact of coding on flow completion time. The lines connect the median and the error bars show the inter-quartile range. [Emulation]

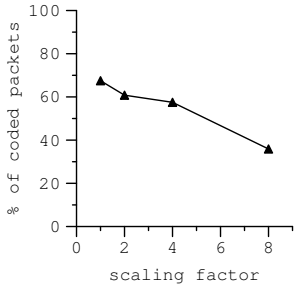


Figure 14: Percentage of coded packets sent as a function of workload intensity. [Deployment]

egy of maximising the expected yield of individual coded packets, ignoring possible optimisations over groups. We do this for simplicity of optimisation and because we do not know at the time of transmission if more can be sent. Our strategy, however, can generate coded packets that are sometimes too complex to be decidable at the other end. At high loss rates, groups of simpler packets can outperform groups of coded packets generated by PluriBus. In our experiment, overestimating loss rate helps because its side-effect is to generate simpler packets. We are currently investigating means for extending Evolution codes to optimise over small groups of coded packets.

6.4.3 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 13 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 very high levels of load. 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. Figure 14 shows the results for experiments using our deployment. As expected, the fraction of coded packets declines as the workload intensifies because there are fewer openings in available bandwidth.

6.5 Delay-based path selection

In this section, we study in detail the behaviour of delay-based path selection of PluriBus.

6.5.1 Benefit of fine-grained striping

We first quantify the advantage of fine-grained packet striping of PluriBus by comparing it with *MinConnPath*. The workload consists of two simultaneous (but not synchronised) TCP flows. Each flow downloads 10 KB of data. A new flow starts when one terminated. We chose this workload because it represents a particularly good case for *MinConnPath*. It enables *MinConnPath* to spread the load evenly on both links by mapping one flow to each link. Such an even spread is 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 the level of individual packets based on the current delay estimate of each link.

In this experiment, we configure the emulator with two links between the VanProxy and LanProxy. Each link has a capacity of 1.5 Mbps and has no loss. 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.

Furthermore, we configure our system to not using any coding (which would, in any case, be superfluous, since there is no packet loss).

Figure 15 shows the flow completion times for the two policies as a function of the difference in the RTT of the two emulated links. We see that as the difference in RTT increases the relative performance advantage of PluriBus increases. In the extreme, when the difference in the RTT is 100 ms, which roughly corresponds to the delay differences of the links in our deployment, PluriBus reduces flow completion time by a factor of 2.

Interestingly, even when the two links have equal delays and *MinConnPath* should lead to almost perfect distribution of connections across them, PluriBus does slightly better. This is because PluriBus exploits the short-term

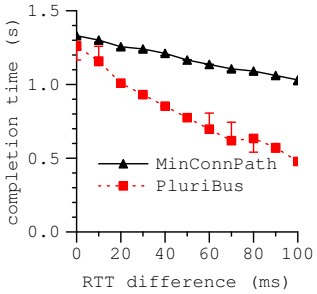


Figure 15: Benefit of delay-based path selection in PluriBus. The lines connect the median flow completion times and error bars represent the inter-quartile range. [Emulation]

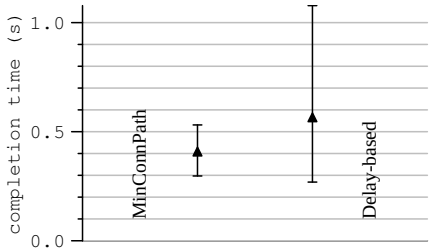


Figure 16: Comparison of MinConnPath and Delay-based striping, on lossy links. [Deployment]

differences in the queue lengths along the two paths that arise when the two TCP flows have different window sizes. Further experimentation shows that this advantage of PluriBus becomes more prominent as transfer sizes increase because that creates bigger differences in the two queue lengths.

These results show that the delay-based, fine-grained striping policy outperforms *MinConnPath* in absence of losses. What happens when losses are present, and no coding is used to mask them? In such a situation, we find that delay-based striping significantly *underperforms* the *MinConPath* policy. This is illustrated in Figure 16. The setup for this experiment is similar to the one used for Figure 8. Indeed, the first bar is same in both figures. The second bar, labelled delay-only was generated by using PluriBus policy, with coding disabled. We see that the fine-grained striping policy performs worse than *MinConnPath*. The reason is that the delay-based striping policy sends most of the data on the WiMax connection that has higher bandwidth, but also has significantly higher loss rate.

We note a somewhat subtle point about this result. In lossy environments, the *MinConnPath* scheme performs better than the delay-based scheme, if no coding is used. Yet, when losses are masked using coding, the delay-based scheme (i.e. PluriBus), performs better. This implies that if the load is so high that there is no spare capacity to send coded packets, PluriBus will underperform *MinConnPath*. We are currently investigating a variation of PluriBus that gracefully switches to using *MinConnPath*, if there is not enough spare capacity to mask losses.

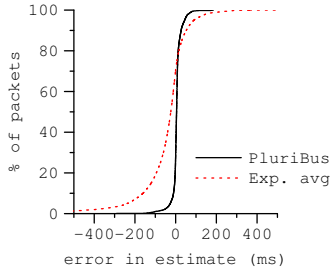


Figure 17: Error in estimated delay along a path. [Deployment]

6.5.2 *Accuracy of path delay estimate*

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

Figure 17 shows the error in our delay estimate. The data in this graph is from the deployment-based experiments of §6.2 and includes all load scaling factors. The curve labelled PluriBus shows that our estimate is highly accurate, with 80% of the packets arriving within 10 ms of the predicted time. This is encouraging, given the inherent variability in the delay of wide-area wireless links (Figure 4).

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. Of these 95% of them 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. This other method estimates path delay by averaging the observed one-way delay that is reported by the other proxy. 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

XX Pablo XX

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) [11], identical delays [36], or ignore losses [15, 32, 30, 10].

A few systems stripe data between end hosts across arbitrary paths by using TCP or a protocol inspired by it along each path [16, 28]. 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 (§6.4).

MAR [32] and Horde [30] 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 [30] 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 [10]. However, the authors did not build a system around the algorithm, nor did they consider the impact of loss.

In contrast to our setting where the wireless links are the bottlenecks, several works focus on the case where the wired links behind the basestations are the bottlenecks and focus on aggregating their bandwidths [22, 9, 38]. These solutions do not directly apply to our case.

Prior work includes studies of TCP's performance over wireless links. Both network-level (e.g., [2, 7, 8] and host-level improvements (e.g., [5]) have been proposed. But neither can be applied to our setting. We do not have access to end hosts. We also do not have access to wireless providers' infrastructure, which network-level methods need, for instance, to quickly react to losses.

Erasure codes [31, 27, 35] and other FEC techniques guard against packet losses. As explained earlier, these techniques do not focus on partial recovery, and are unsuitable for our environment. Systems such as MORE [6] and COPE [23] use network coding in multi-hop wireless mesh networks. This is a very different context than ours; these systems exploit the broadcast nature of wireless medium and code across multiple nodes.

Our Evolution codes are inspired by Growth codes [21]. Growth codes were designed to preserve data in large sensor networks with failing sensors. This is a very different application domain, and many of the assumptions made in the design of Growth codes are specific to that domain. For example, they assume that the receiver starts out with no information also assume that multiple senders, each with a different data set, are attempting to communicate with a single receiver.

An unconventional aspect of our design is that we strive to use all spare capacity in the system, without worrying about efficiency. In a recent position paper [29], we discussed broadly the value of such an approach. We argued that the approach is counter to common design practices. It can be valuable in many settings but 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.

Split TCP [19, 24] is another approach to improve TCP's performance on paths involving wireless links. In this approach, one or more proxies are used to break the end-to-end TCP connection into multiple, independent segments, each running its own TCP connection. Split TCP improves performance by performing loss recovery independently on each segment, without waiting for end-to-end feedback. This approach works well in scenarios where lossy segments have have low round trip delay; for example; when the last-hop wireless link on an Internet path is inducing significant loss. However, the Split TCP approach is not suitable in our setting for two reasons. First, it violates the end-to-end semantics of the TCP connection, and hence it can not be used in in presence of IPSec. Many corporate networks use IPSec-based VPNs for remote access, and the Split TCP approach can not be used for these VPNs. Second, since we don't have access to the internal network of our wireless carriers, we can only split the TCP connections at the VanProxy and the LanProxy. While the segment between the VanProxy and the LanProxy is indeed the most lossy segment on most end-to-end paths, it is also responsible for the majority of the end-to-end RTT. Hence, the Split TCP does not offer any advantage over direct TCP connection. We have verified this by building a Split TCP implementation. We omit these results due to lack of space.

Another possibility is to use the TCP-in-TCP encapsulation. This is similar to the retransmission approach discussed in Section 6.

8. DISCUSSION

The aggressive addition of redundancy in PluriBus is

based on a selfish perspective. Bus operators typically subscribe to a fixed-price, unlimited usage plan with their wide-area wireless provider. 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. There are two defenses against an exorbitant increase in operational cost. 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 believe 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.

9. CONCLUSIONS

We designed and deployed PluriBus, a system to provide high-performance Internet connectivity on-board moving vehicles. PluriBus seamlessly combines multiple, heterogeneous wide-area wireless links into a single, reliable communication channel. Our evaluation shows that it reduces the median flow completion time by a factor of 2.5 for realistic workloads.

The key technique in PluriBus is *opportunistic erasure coding* in which coded packets are sent only when there is instantaneous spare capacity along a path. We show how this can be accomplished in a way that does not hurt data traffic. While we use this capacity to send coded packets, the underlying mechanism is general and can be used in many settings, including non-vehicular ones. We are currently building a background trasfer service using this mechanism to upload logs from our buses.

In contrast to full recovery, which is the focus of most existing erasure codes, our *Evolution codes* are designed for partial recovery. They greedily maximize the expected yield of each coded packet by explicitly taking into account what might already be present at the receiver. Because full recovery is not necessary in many scenarios, we believe that Evolution codes are broadly useful.**XX examples? XX**

10. REFERENCES

[1] S. Alexander and R. Droms. DHCP options and bootp vendor extensions, 1997.

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

[3] M. Balakrishnan, T. Marian, K. Birman, H. Weatherspoon, and E. Vollset. Maelstrom: Transparent error correction for lambda networks. In *NSDI*, Apr. 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] R. Chandra, P. Bahl, and P. Bahl. MultiNet: Connecting to multiple IEEE 802.11 networks using a single wireless card. In *INFOCOM*, Mar. 2004.

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

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

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

[13] Feeney wireless. www.feeneywireless.com.

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

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

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

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

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

[19] R. Jain and T. Ott. Design and implemenation of Split TCP in the Linux Kernel. In *GlobeComm*, 2006.

[20] Junxion's cell through. http://www.unstrung.com/document.asp?doc_id=94452.

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

[22] S. Kandula, K. C.-J. Lin, T. Badirkhanli, and D. Katabi. FatVAP: Aggregating AP Backhaul Capacity to Maximize Throughput. In *NSDI*, April 2008.

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

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

[25] Kyocera mobile router. http://www.kyocera-wireless.com/kr1-router/.

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

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

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

[29] 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.

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

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

[32] P. Rodriguez, R. Chakravorty, J. Chesterfield, I. Pratt, and

S. Banerjee. MAR: A commuter router infrastructure for the mobile Internet. In *MobiSys*, June 2004.
[33] S. Sanghavi. Intermediate performance of rateless codes. In *Information Theory Workshop*, 2007.
[34] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The end-to-end effects of Internet path selection. In *SIGCOMM*, Aug. 1999.
[35] A. Shokrollahi. Raptor Codes. *ToIT*, 52(6), 2006.
[36] A. Snoeren. Adaptive inverse multiplexing for wide-area wireless networks. In *Globecom*, Dec. 1999.
[37] N. Spring, R. Mahajan, and T. Anderson. Quantifying the causes of path inflation. In *SIGCOMM*, Aug. 2003.
[38] N. Thompson, G. He, and H. Luo. Flow scheduling for end-host multihoming. In *INFOCOM*, 2006.
[39] Top Global USA, Inc. www.topglobalusa.com.
[40] More cities offer Wi-Fi on buses. http://www.usatoday.com/tech/wireless/2008-04-10-wifi_N.htm.
[41] Amtrak and T-Mobile offer WiFi at select stations, sounder trains have free WiFi. http://blogs.zdnet.com/mobile-gadgeteer/?p=642.
[42] WiFi incompatibility on Seattle metro? http://www.internettablettalk.com/forums/archive/index.php?t-18539.html.
[43] The last fifty feet are WiFi. Reader comments at http://www.bwianews.com/2007/04/the_last_fifty_.html.
[44] Sound transit riders. A mailing list for Microsoft employees.
[45] Google's buses help its workers beat the rush. http://www.nytimes.com/2007/03/10/technology/10google.html.
[46] Metro bus riders test county's first rolling WiFi hotspot. http://www.metrokc.gov/kcdot/news/2005/nr050907_wifi.htm.
[47] 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.
[48] 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/.
[49] Free Wi-Fi on Sound Transit. http://www.soundtransit.org/x6083.xml.