Data Management Mechanisms for Embedded System Gateways

Justin Ray
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
justinr2@cmu.edu

Philip Koopman
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
koopman@cmu.edu

Abstract

It is becoming increasingly common to connect traditional embedded system networks to the Internet for remote monitoring, high-level control and integration. It is necessary to protect each part of the interconnected system from faults and attacks which propagate from the other side. One architectural approach is to add a gateway to the embedded system to receive Internet traffic and disperse data to the embedded system, but there is no clear recipe for building such gateways. Since Internet routers commonly use queues to manage traffic, we examine the effectiveness of queues for the embedded system gateway domain. We perform a series of experiments to evaluate the effectiveness of the queue mechanism and various queue management techniques. We show that queues can exhibit poor performance in the context of real-time embedded system gateways due to problems with message latency and dropped messages. We then introduce the concept of a filter mechanism and show that a simple filter mechanism can outperform queue mechanisms when used in the gateway to manage real-time state-oriented data streams.

1 Introduction

Embedded systems that traditionally have operated in isolation or on closed networks are being connected to the Internet or to other networked systems to increase functionality and consolidate operations. An embedded system gateway is the device that manages the flow of information between these two networks, and we will use the term "gateway" throughout this paper to refer to such devices.

In order for the system to be survivable, the gateway must control the flow of information in a way that meets the requirements of the system and does not adversely affect local traffic on the individual networks [6]. Because these networks have different timing properties, normal (non-faulty) timing of message arrival on the Internet side may cause the gateway to fail to meet real-time requirements on the embedded side. Standard security and survivability practices, while important, are not geared toward protecting systems against these kinds of faults. Gateways must be designed to provide isolation against timing variations and faults between the two networks in order to maintain the dependability of the interconnected systems, but there is very little guidance in the literature about how to do so. This paper presents an initial investigation of the mechanisms that can be used in a gateway for timing fault containment, which is a first step toward developing survivable gateways.

A gateway is conceptually similar to routers used in Internet infrastructure. Since queues are commonly used to manage packet flows in Internet routers, we begin by using queues as data management mechanisms in gateways. We explore various design parameters for queues, including queue length and queue management policies.

In order to evaluate the performance of data management mechanisms in the gateway, we developed a simulation framework which models the gateway and the timing characteristics of the two networks attached to it. Using this simulation, we are able to study and compare different mechanisms under repeatable conditions. We use this framework to model the scenario of periodic state variable data being transmitted from a server on an enterprise network via a gateway to an embedded control system on an embedded network.

We begin by exploring the design space of queuing mechanisms and evaluating the effectiveness with which queues with various parameters can mitigate timing differences. Based on our experimental results, we will show that queue mechanisms do not perform well in this scenario because of the latency they introduce and because of dropped messages. Analysis of the queue mechanism yields an important insight: queue underflow is a significant source of delay because it results in lost bandwidth for messages delivered in

Preprint from 2009 International Conference on Dependable Systems and Networks (DSN09)

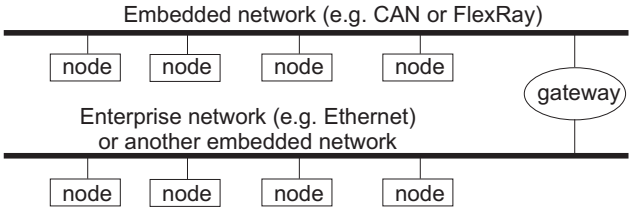

Figure 1. A basic configuration of two networks and a gateway

the real-time network. Based on this analysis, we propose a filter mechanism to overcome the shortcomings of queue mechanisms.

The remainder of this paper is organized as follows: Section 2 expands on the concept of an embedded system gateway and describes the constraints under which these systems operate. Section 3 highlights previous work related to this paper. Section 4 describes queue mechanisms and their design parameters. Section 5 describes our experimental approach, and Section 6 presents the results of our evaluation of queue mechanisms. Section 7 offers insight into the shortcomings of the queue mechanisms and how those shortcomings can be mitigated. Section 8 defines filter mechanisms, and Section 9 offers experimental results comparing filters and queues. Section 10 summarizes our contributions and outlines future directions for this research.

2 Background

A gateway is a system entity that manages the flow of information between two networks. Figure 1 shows an example of two networks interconnected by a gateway. To implement communication between two networks, there must be a device with interfaces on both networks to effect message passing between the networks. In addition to the physical interfaces, the gateway also must determine which messages are forwarded, if any translation or repackaging will take place, and when to send the messages based on the network schedules. While all of the gateway logic could be included in such a dual-homed device, it is also possible to allocate some of the decision making behavior to individual nodes on the two networks. Therefore, the term "gateway" as we will use it in this paper refers to the abstract concept that includes not only a dual-homed device, but also the logic governing gateway operation, regardless of where that logic is located.

Embedded system gateways go beyond traditional routers because they connect two different types of networks. Different networks can have different timing requirements because of the different methods they use for access control and error correction. For example, TCP/IP over Ethernet uses retransmission mechanisms to ensure reliable delivery, but packets have highly variable latency which is unbounded in the worst case. A Controller Area Network (CAN) has bit-arbitrated message prioritization, so the latency of any message is affected by the schedule of all higher priority messages [10]. FlexRay [2] and the Time Triggered Protocol (TTP) [13] are time-division multiplex access (TDMA) networks that execute periodic schedules and make strict real-time guarantees for periodic schedules.

There are several scenarios where a gateway might be used. The requirements for the gateway will likely be different in each scenario. A gateway in a real system might be responsible for several data flows which could encompass more than one of these scenarios.

- *Enterprise-to-Embedded*—data is sent to the gateway over enterprise networks (such as the Internet or a LAN). The gateway then sends this data out on a real-time embedded network. An example of this system is a supervisory control application that runs on a corporate LAN and connects via gateway to a factory control network.
- *Embedded-to-Enterprise*—the gateway receives data from a real-time embedded network and then sends it to a server or personal computer over an enterprise network. This scenario might arise if an embedded system (such as a thermostat on a home automation network) reports its status to an Internet server.
- *Embedded-to-Embedded*—the gateway connects two embedded networks that are of a different type. The two networks could also be of the same type but configured to run at different speeds or with different schedules. For example, many automobiles have several embedded networks. These networks could be interconnected to implement new system features.

A *data management mechanism* is the part of the gateway that handles the transfer of data from the input interface to the output interface. When we refer to a mechanism in this paper, we are referring to these data management mechanisms. The mechanism determines the order in which messages are sent out and which messages (if any) are dropped. It also encompasses whatever storage is required for pending messages. A mechanism only handles messages of one type, such as messages related to one state variable. If there are multiple data streams in the gateway, each stream has its own data management mechanism.

At a high level, a gateway performs a task similar to that of Internet routers. These routers commonly use queue mechanisms to manage packet flows, so we begin our examination of data management mechanisms with queue mechanisms.

The motivating example which we will examine in this paper is an enterprise-to-embedded scenario. An incoming stream of vehicle speed data arrives at the gateway via an enterprise network (e.g. wireless or cellular network). The messages are sent out by the gateway during a periodic time slot on an embedded control network (e.g. TTP, FlexRay or CAN with rate monotonic scheduling). In our simulation, we model message arrivals with a Poisson random process and message departures with a deterministic periodic process. Our example has single data flow and a single data management mechanism to handle it.

In addition to the example used in this paper, we believe these techniques can be applied to any system that a) uses real-time control or monitors state variables and b) has components that are sufficiently remote, plentiful or mobile that the use of existing network infrastructures is desirable or necessary to make the system economically feasible. Most Supervisory Control and Data Acquisition (SCADA) systems, such as pipeline and power grid control systems, would fit this description. Although current SCADA applications use remote connections primarily for monitoring activities, the expansion of these systems to include real-time control seems likely in the future.

We believe the data sets we used in our experiments are representative of some types of data from these applications. While applications that involve data with a fundamentally different character might produce different results, the important insights from our experiments are the identification of queue underflow as a major cause of queue delay and the proposed solution (filters) for mitigating this problem. For other types of data, it is likely that there are other mechanisms which can be used in the gateway to meet application requirements. By continuing to examine likely scenarios and use cases for embedded system gateways in the future, we can identify and develop additional mechanisms to meet a wide variety of system requirements.

3 Related work

There are numerous network devices which are referred to as gateways, especially in the Internet domain, but we have found few examples that deal with connections to embedded networks. One design for a gateway between Internet and CAN is presented in [4], although that work focuses on the hardware design and implementation. The timing analysis focuses on the capability of the gateway to process all the messages and assumes that the CAN network is strictly an event-triggered network. It does not address the situation of how to handle arriving messages if they cannot immediately be sent out on the CAN network, as might be the case if transmission is blocked by higher priority messages.

Queuing is the primary mechanism used to manage packets in Internet routers, and much work has been done with active queue management techniques for optimizing throughput and implementing congestion control algorithms. Random Early Detect (RED) [3] or BLUE [1] are two examples that are designed to maximize throughput and reduce congestion. These techniques are usually applied to unbounded queues. [12] modifies RED in an attempt to maintain a target queue length, but the proposed method provides only a probabilistic bound.

Although these techniques are quite successful in the Internet domain, they are intended to optimize bandwidth utilization and throughput. In contrast, an embedded system gateway should be designed to provide for the timeliness of the data that is passed through the gateway. This implies some fundamental differences in the operation of an embedded system gateway. Active queue management techniques in Internet routers use dropped packets as in implicit communication channel to control the sender rate. When a packet is dropped, the TCP retransmission mechanisms cause the packet to be resent, so the dropped packet will eventually be delivered. In contrast, our gateway acknowledges every message and then delivers the message to the data management mechanism. If the mechanism indicates that a message should be dropped, then the message is dropped silently and no notification is given to either the sending or the receiving network. This approach is useful for two reasons. First, for the particular case of TCP/IP, it avoids the delay of another round-trip time to resend the packet on the Internet side of the gateway. Second, it allows us to apply the same techniques to networks which do not have an automatic retransmission mechanism.

4 Queue mechanisms

Since queue mechanisms are applied successfully in Internet routers, we begin our examination of data management mechanisms by evaluating their effectiveness as mechanisms for embedded system gateways. All the queues discussed in this paper use the first-in-first-out (FIFO) queue discipline.

There are two key parameters for queue design that we consider: queue length and queue management policies. For queue length, we consider both bounded queues, which are constrained to a certain maximum length, and unbounded queues, which are allowed to grow to any length. While it is not possible to implement a truly unbounded queue, we assume for the sake of analysis that the storage capacity of any practi-

cal implementation can be made arbitrarily large. For queue management policies, we consider behavior during underflow and overflow situations.

4.1 Queue underflow policies

Queue underflow is one exceptional situation that queue management policies must deal with. Queue underflow occurs when the outgoing network schedule is ready for a new message, but the queue is exhausted and no new message has arrived. This situation is likely to arise if the arriving data is bursty (e.g. a Poisson random process), but it could also occur if both the incoming and outgoing networks are periodic with random jitter.

We refer to the underflow policy we use as a mailbox policy because of its similarity to the mailbox implementation used in CAN controllers [10]. The mailbox holds the most recently transmitted value, and if no new value is available from the queue, that same value is sent again. Depending on the implementation, a staleness indicator may also accompany the repeated value.

There are two alternatives to the mailbox policy: send an invalid value or send no value. If an invalid value is sent or no value is sent, the application is going to continue to use the last valid value (e.g. the set point for an actuator remains at the last received value), so the net effect of these other policies is very similar to that of the mailbox policy.

If the data being sent is event oriented data, then the mailbox policy could result in the system interpreting repeated values as additional events. In this case, the null or invalid message policies might be preferred. Since the examples discussed in this paper are all concerned with periodic state variable data, all the queue mechanisms discussed use the mailbox policy.

4.2 Queue overflow policies

Queue overflow policies describe the action to be taken when the queue exceeds its designed maximum length. These policies only apply to bounded queues. For an unbounded queue, an overflow condition cannot occur unless the physical limitations of the system are exceeded. Some policies may drop more than one message or cause the *incoming* message to be dropped. As was mentioned in Section 3, when a message is dropped by the gateway, the fact that the message was dropped is not reported to either the sending or receiving network.

We have identified four queue overflow policies which we describe below.

The *Drop Newest Policy* requires that the newest message (the arriving message) be dropped. This is similar to the active queue management technique known as Drop Tail [3], which has been used in Internet routers.

The *Drop Oldest Policy* requires that the oldest message (i.e. the message at the head of the queue) be dropped. This technique is more useful for state-oriented messages where the more recent messages contain a more accurate description of the current system state. This is similar to the Drop Front congestion control technique proposed in [7].

The *Drop Random Policy* requires that a message be dropped at random from the queue when an incoming message arrives at a full queue. The incoming message is included in the pool of candidate messages to be dropped. This technique is similar to the Random Early Drop technique [8].

The *Drop All Policy* requires that the queue be flushed (completely emptied) when a new message arrives at a full queue. The arriving message is not dropped, but all the messages already stored in the queue are dropped.

While the Drop All policy is novel, the remaining three are an application of existing queue management techniques to embedded system gateway queue mechanisms.

5 Experimental approach

Here we describe the experimental setup we used to evaluate the various queue mechanisms, the apparatus we used to collect input data for the simulations, and the metrics we recorded to evaluate and compare different mechanisms.

5.1 Simulation framework

In order to evaluate the performance of the various queue management mechanisms, we developed a discrete time event simulator in Java. The simulation models an arrival process which delivers data to the queue and a service process which removes messages from the queue. Both processes can be specified to be deterministic (e.g. periodic) or random according to a probability distribution.

All the random elements or sequences in the simulation are generated using a deterministic pseudo-random number generator. The software can repeatably generate the same pseudo-random sequence from a given seed value. Thus, the same pseudo-random arrival sequence can be recreated and applied to gateways which implement different data management mechanisms to allow for a fair comparison of their performance.

5.2 Input data collection

In these experiments, we are concerned with state-oriented periodic data streams. We developed a data

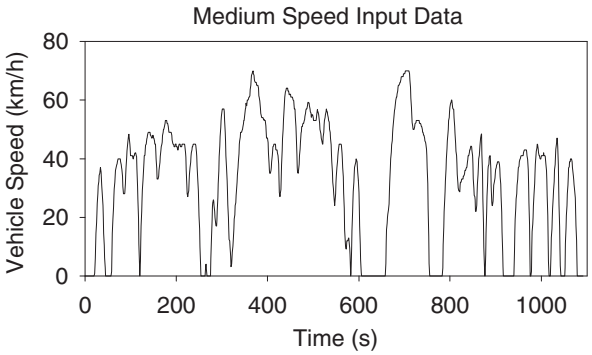

Figure 2. One input data set for the simulation, collected on residential roads with relatively few stop signs and stop lights.

collection system that uses the automotive standard OBD-II diagnostic interface [11] to record the speed of a vehicle during operation. The resulting data is a reasonable example of a state-oriented data set. An example of an application which might generate similar data is a traffic control system that reports the speed of vehicles further up the road via roadside transmitters or ad hoc vehicle-to-vehicle networks. This data could then be delivered via real-time network to an onboard embedded controller and used to make adjustments to adaptive cruise control settings.

We collected four different data sets to use as inputs to the simulation. The first data set is a neighborhood driving scenario with low speeds and frequent stops and starts. The second set (shown in Figure 2) is a medium speed scenario on roads that are not highways, but have few stop signs or stop lights. The final two sets are highway driving in light traffic and in heavy traffic.

5.3 Metrics

There are several relevant metrics which were recorded in the simulations. Each metric is aggregated over all values for a single trial, so it produces a single value for each trial. In the results, we compare the distribution of these values from experiments with different gateway configurations (e.g. different mechanisms).

- *Maximum queue length* is the maximum queue length observed during a single trial.
- *Average queue delay* is the delay for each message (time between arrival at the queue and departure from the queue) averaged for all messages in a single trial. Dropped messages are not factored into this metric, and neither are duplicate deliveries that occur because of the mailbox policy.
- *Dropped message count* is the total number of dropped messages in a single trial.
- *Mean value error* is a metric designed to capture how well a data management mechanism preserves the data sequence. It is computed by recording the point-by-point difference between the original data sequence and the data sequence output by the gateway. The average of the absolute value of these differences is computed over the whole run.

5.4 Experimental setup

Our experiments are designed to model the enterprise-to-embedded scenario described in Section 2. Internet traffic has been shown to be bursty [5], so we choose to model the arriving data with a Poisson random process. [9] discusses several delay models that can be used to model the arrival process: constant delays, independent random delays (e.g. the Poisson process), and Markov chain models (which capture the effect of network load on the distribution of delays). The Poisson process model is simpler because it does not require an exploration of the additional parameters of the Markov chain models, but it captures the bursty nature of the arriving data. We believe that experiments with other bursty arrival process would have similar results.

For each experiment, the simulator was configured with a Poisson arrival process with mean interarrival time of one message per second and a periodic service process also with a period of one second. We use one of the vehicle speed data sets described in Section 5.2 as input data. The simulated gateway is configured with a particular data management mechanism (e.g. finite queue of length 50 using the Drop Oldest overflow policy). A trial is a single run of the simulation and produces a single value for each of the metrics described in Section 5.3. In each trial, a different pseudo-random arrival sequence is applied to the gateway. Each experiment consists of 5000 trials. The sequence of data *values* delivered to the gateway in each trial is the same (such as those pictured in Figure 2). Only the *timing* of the arrivals changes from trial to trial. An experiment set is a series of experiments performed with different mechanisms (e.g. a set of experiments on queues of length 10, 20, and 50) intended to compare the performance of the mechanisms with respect to one or more metrics. Experiment sets are repeated using each of the four data sets as inputs.

6 Queuing Results

This section describes the queue mechanisms that were used in the various experiments and highlights some of the significant results. Although experiments were performed on all four of the input data sets described in Section 5.2, we have chosen to report only

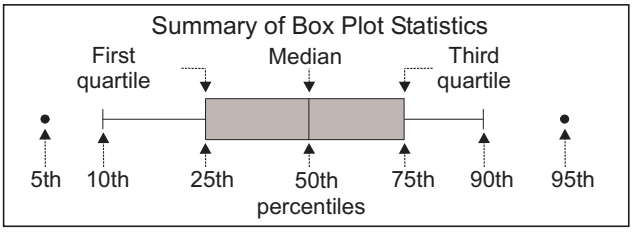

Figure 3. Summary of the statistics provided by the box plot diagram.

the results from the second data set (shown in Figure 2) because of the limited space available. The results from the other data sets are qualitatively similar and also support our conclusions.

Many of the results presented below use box plots to summarize the results for a particular metric for a single experiment. The statistics given in a box plot are summarized in Figure 3. The results from a set of experiments are presented in a single graph to facilitate comparison of the results.

6.1 Unbounded queues

Our first experiment used an unbounded queue mechanism. Figure 4 shows a selection of time series data from a single trial of the experiment. Part (a) shows the input and output data streams and highlights the delay between the input and output. Part (b) shows the size of the queue over time. The delay increases as the queue length increases. As might be expected, the delay is directly proportional to the queue length, since the length of the queue when a message arrives determines how long it remains in the queue.

One goal was to see how long the queues could grow. Figure 5 shows the distribution of maximum queue lengths for this experiment. While the median of the distribution is around 38, the maximum queue length observed was 125. Although longer queues are less likely, there is no theoretical upper bound on the worst case queue length for an infinitely long data set.

The mean value error results for unbounded queues experiments are included in Figures 8 where they are compared to the results from the bounded queue experiments described in Section 6.2.

These experiments show that transient queue lengths and delays can grow quite large, even if the average rate of the data going in and out of the queue is the same. While this result may be expected, it is important because it leads us to examine bounded queues as a way to mitigate this delay.

6.2 Bounded queues

We now examine bounded queues, since a bounded queue should have bounded delay. When we study

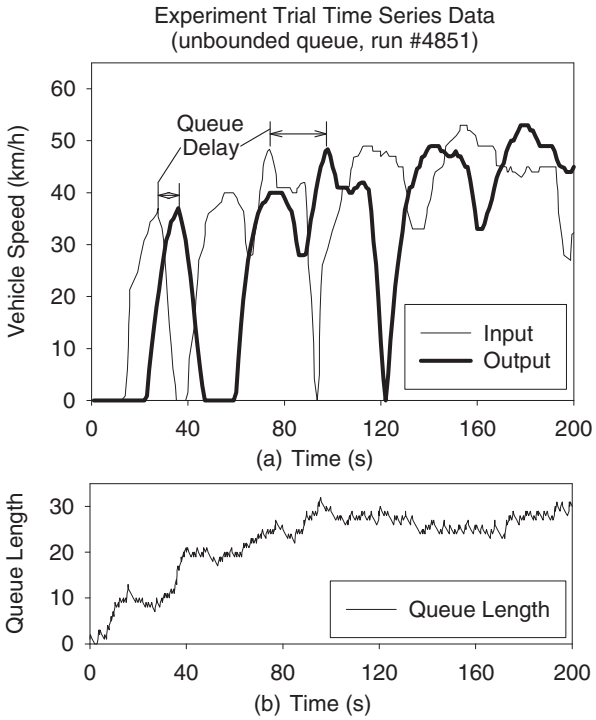

Figure 4. Time series data from a single trial. Queue delay increases as the queue length increases.

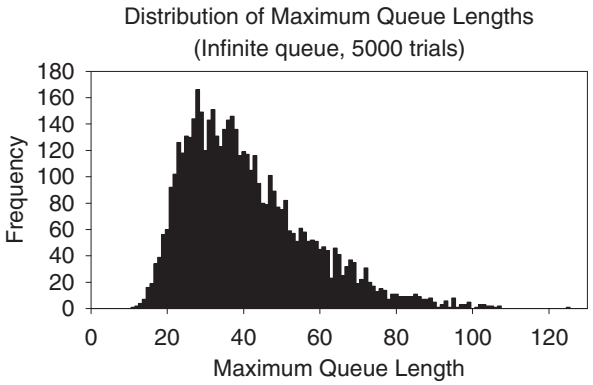

Figure 5. Distribution of maximum queue lengths observed during each of 5000 trials.

bounded queues, we consider two parameters: the length of the queue and the overflow policy.

First, we examine the effect of queue length on delay and on the number of dropped messages. Figure 6 shows that the number of dropped messages decreases as the queue length increases. For queues of length 50 or more, very few messages are dropped at all. This is because longer queues are less likely to overflow.

On the other hand, Figure 7 shows that the average delay increases as the queue size increases. For queues

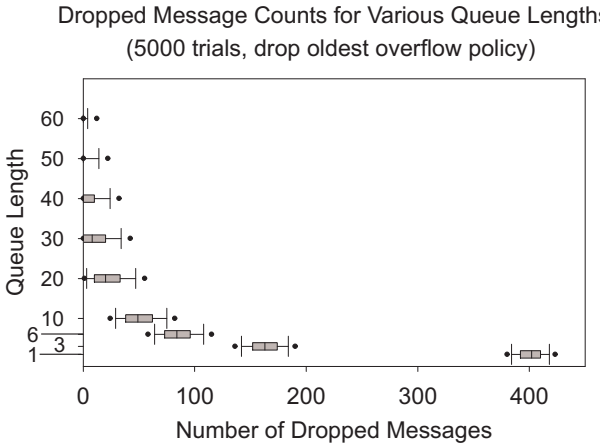

Figure 6. Summary of the total number of dropped messages from each trial for experiments with queues of various lengths.

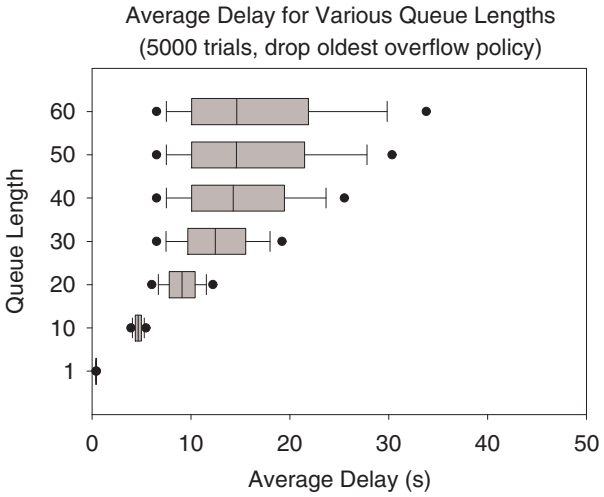

Figure 7. Summary of the average queue delay for queues of various lengths.

of length 40 or longer, the median value of the average delay begins to level off, although upper bound on delay continues to grow. Just as longer queues are less likely to overflow, they are also less likely to be full, which means that as queue bounds become larger, the median delay is governed less by the length of the queue and more by the timing of the arrival messages (recall that each experiment uses the same set of arrival sequences).

Based on the results in Figures 6 and 7, we observe that there is a trade off between the number of dropped messages and the average queue delay.

Now we examine the effect of queue length on the mean value error. Recall that the mean value error metric produces a single value for each trial in the experiment. The box plots in Figure 8 compare the mean

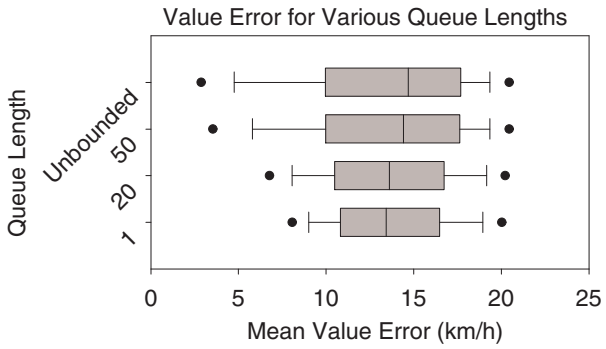

Figure 8. Box plot summary of mean value error for various queue lengths. Shorter queue lengths have a slightly lower median value.

value error for experiments run with bounded queues of length 1, 20 and 50 and an unbounded queue. For the bounded queues, the Drop Oldest overflow policy was used.

We might expect that reducing queue delay would also decrease the mean value error in the output. Indeed, the results in Figure 8 show that median error does go down slightly as the queue length is reduced. However, we also observe that 5^{th} percentile of the error actually increases for short queues. Although we have reduced the delay by reducing the queue size, the shorter queues drop more messages, and these dropped messages also contribute to the mean value error. So, although we can use the queue length parameter to choose a point in the trade off space between dropped messages and average delay, the queue length parameter has very little effect on the mean value error.

Figure 9 shows a comparison of experiments using a length 50 bounded queue with various overflow policies. The performance of the Drop Newest, Drop Oldest, and Drop Random policies is almost the same. This is because when an overflow condition occurs, each policy drops a single value. Although each policy selects a different message to drop, the number of dropped messages, and thus the overall effect, is relatively small. The only policy that exhibits different behavior is the Drop All policy. The performance of this policy is worse because the flushing of the queue results in a large number of dropped messages.

The insight to be gained from these experiments is that neither queue length nor overflow policy can significantly improve the mean value error performance of the gateway.

7 Analysis

As the results in Section 6 show, the delay introduced by the queue can increase the error in the values

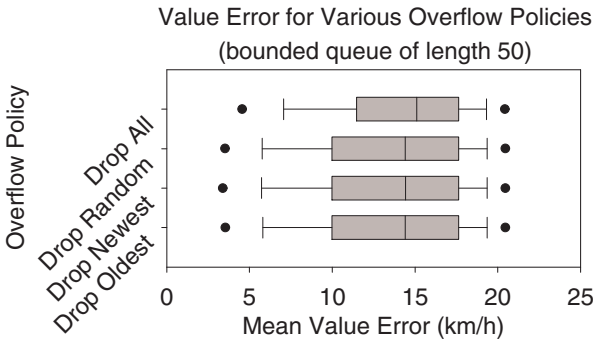

Figure 9. Box plot summary of mean value error for queues employing various overflow policies. There is very little difference in the performance of the policies.

output by the gateway. A detailed examination of the queue in operation can offer insight into how queue delay arises. This, in turn, offers insight into how to mitigate delay.

It turns out that queue underflow is the root cause of message delivery delay. In the example we consider, the average input and output rates of the gateway are the same, so in the steady state, there should be no accumulation of messages in the queue. However, the experimental results show that this is not the case.

Figure 10 sketches the way that queue underflow can introduce delay into the queue. Part (a) of the figure shows the ideal case, when each incoming message arrives in time for its time slot in the output stream. No messages accumulate in the queue and there is no delay in the output stream values.

The arrival process that we use to model enterprise networks is a Poisson process, so the traffic arrivals are bursty, as pictured in part (b) of the figure. When the first three messages arrive in a burst, they are queued and delivered in their appropriate time slots. Because of the long quiescent period between the bursts, the fourth message has not arrived when the fourth time slot comes up at the output, so the third message is sent again (per the mailbox policy). When the fourth message does arrive, it is sent in the time slot where the fifth message should go, and the fifth message is delivered in the sixth slot, and so on. The time slot missed by the fourth message cannot be recovered, and the steady state size of the queue has increased by one, adding one message period delay to each delivered message. This process can happen repeatedly, causing the queue to grow longer and longer as the result of either normal timing variations or a malicious attacker purposely clumping message arrival times.

The solution that we propose to the problem of queue underflow is shown in Figure 10(c). When under-

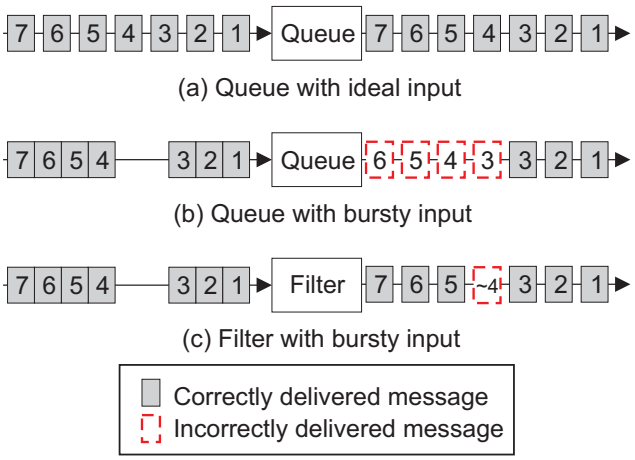

Figure 10. Comparison of Queues and Filters with bursty inputs

flow occurs and the fourth message is not available in the fourth time slot, the data management mechanism produces an estimate of the missing data. When the fourth value arrives, it is *not* delivered in the fifth time slot. In fact, it is not delivered at all. This eliminates the delay caused by the missed time slot.

8 Filter mechanisms

A mechanism that implements the behavior shown in Figure 10(c) can be thought of as a queue with a special policy that drops an incoming message for every duplicate message that has been sent. This policy is not an *overflow* policy because the messages are not dropped based on the length of the queue. If this special queue mechanism uses the mailbox policy for underflow, then the estimate for the missing message will be the last sent message. This queue mechanism can intelligently drop messages in order to reduce delay, but since the queue mechanism can only process messages based on their arrival order and timing, the estimates for the missed message values cannot be more sophisticated than those generated by the mailbox policy.

We propose to generalize the idea of using previous data values to estimate missing output values. We call a mechanism that looks at message values as well as message timing and arrival order a *filter*. Additionally, filter mechanisms may aggregate or modify message values. One consequence of this definition is that a queue may only deliver a message value that it has received in the incoming message stream, but filters could potentially deliver a message with a value that has never been seen at the input of the gateway, although this new value may be a function of previous input values. A wide variety of models can be applied to the data to generate an estimate. For example, the

model could predict a missing value by taking the average of the three previous values or by performing regression analysis on the previous data and extrapolating to compute the new value. The data model should be chosen to fit the data that is being filtered.

Although there is a very large design space for data models that can be used in these filters, our purpose here is simply to demonstrate that filters can be more effective than queues in the enterprise-to-embedded scenario. Therefore we have chosen the mechanism that is described in Figure 10(c). This mechanism uses a simple data model which approximates missing values by zero order (constant) approximation. In other words, the missing value is approximated by the last value that was received at the gateway input. As we have noted, this mechanism is on the boundary between queues and filters, and could be classified as either one. The comparison in Section 9 below shows that this mechanism exhibits significantly better data value error performance. We believe that further improvement could be obtained by using more sophisticated data models, and for this reason we choose to classify this mechanism as a filter mechanism.

9 Comparing filters and queues

To compare the filter and queue mechanisms, we evaluate them using the mean value error metric. Recall from the results in Section 6 that the queue length and overflow policy parameters had little impact on the mean value error metric. Figure 11 shows the mean value error summary comparing the filter mechanism to length 1 and 50 bounded queues and an unbounded queue. The bounded queues used the Drop Oldest overflow policy. Per the results in Figure 9, other overflow policies would yield similar results. Although the 95^{th} percentile of the mean value error is only slightly lower, the filter mechanism shows a significant bias toward lower mean value errors. To highlight this improvement, the same results are presented in Figure 12 in the form of cumulative distribution functions.

These results are not meant to imply that queue mechanisms have no place in embedded system gateways. There may be other scenarios with different requirements that are more suited to their characteristics. These results do demonstrate that a further exploration of filter mechanisms is warranted.

10 Conclusions and future work

As it becomes more common to network and internetwork embedded systems, there is a need to develop mechanisms and policies for embedded system gateways and to develop guidance on how these mechanisms and policies can be used. We have begun this work by

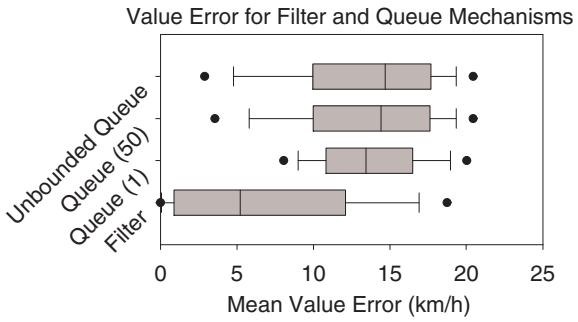

Figure 11. The filter mechanism shows a significant improvement in mean value error over queue mechanisms.

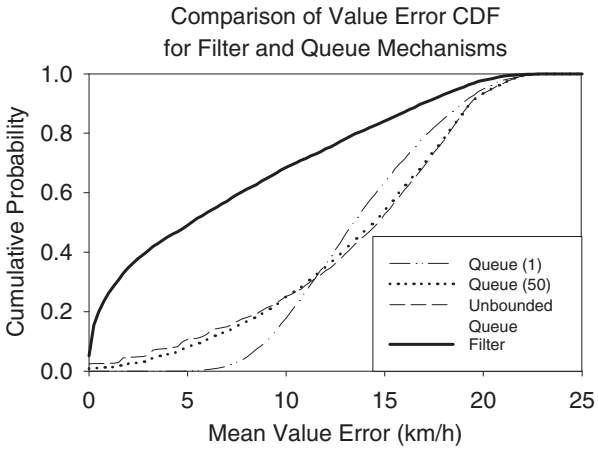

Figure 12. The CDF of the filter mechanism shows a significant bias toward lower errors compared with the queue mechanisms.

evaluating queue mechanisms and by proposing instead using a filter mechanism.

Although queues are used successfully in Internet routers to manage bandwidth and optimize throughput, we have shown that queue mechanisms are not well suited to at least some embedded system gateway applications which require timely delivery of data. An important insight is that queue delay is a result of queue underflow. Our evaluation has demonstrated the inherent trade off between dropped messages and delay, and we have shown how both delay and dropped messages can introduce error in periodic, continuous-valued state variable data streams.

We have also demonstrated that a simple filter mechanism can mitigate the shortcomings of the queue mechanism and improve gateway performance over queue mechanisms.

One assumption in our approach to data handling in the gateway is that the minimum sampling speed

for the application can be met by the gateway mechanisms. If timing differences in the networks result in too many messages being dropped, then using that network topology may be infeasible. It is possible that mechanisms which include estimation (such as more advanced versions of the filters proposed here) can mitigate this shortcoming, but it depends on the character of the data and the application requirements.

We believe that, in contrast to Internet routers, successful embedded system gateway designs are going to require some application knowledge, or at the very least, knowledge of the requirements for a particular data stream. If there are multiple data streams in the gateway, then some method (such as *a priori* configuration) must be in place to enumerate these streams, to select a data management mechanism for each stream, and to define parameters (such as queue length for a queue mechanism or data model type for a filter mechanism).

By enumerating various mechanisms and the types of scenarios they are suited for, we can provide a tool kit of data management mechanisms that can be used at design time to develop gateway implementations with a fixed set of mechanisms for specific applications and data types. Given sufficient understanding of the impact different mechanisms have on various types of data streams, applications themselves could communicate with the gateway at runtime and request a particular mechanism for a particular data stream via an API, similar to the plug-and-play (uPNP) protocol employed in many commodity home routers. In this way, the embedded system gateway can move toward becoming a generic device that can be used with many systems and applications.

As we continue to study the design and implementation of embedded system gateways, there are several avenues that remain to be explored. The simple filter introduced in this paper is just the beginning of the design space for filter mechanisms. We plan to examine various estimation and regression methods and evaluate their effectiveness as data models for various types of state variable data. We will also continue to identify and evaluate new data management mechanisms. For example, one mechanism that is likely to be needed is a mechanism to aggregate data. Such a mechanism could be used when connecting a high bandwidth network to a low bandwidth network. We also plan to evaluate the mechanisms we have already identified under other scenarios, such as the embedded-to-enterprise and embedded-to-embedded scenarios. Just as the evaluation presented here gave insight into the design of the filter mechanism, we expect these other scenarios to yield similar insights and possibly new mechanisms as well.

Acknowledgment

This research was funded by General Motors through the GM-Carnegie Mellon Vehicular Information Technology Collaborative Research Lab.

References

[1] W. C. Feng, K. Shin, D. Kandlur, and D. Saha. The BLUE active queue management algorithms. *Networking, IEEE/ACM Transactions on*, 10(4):513–528, Aug 2002.

[2] FlexRay-Consortium. FlexRay communications system, protocol specification, version 2.0. Request online: `http://www.flexray.com/specification_request.php`.

[3] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *Networking, IEEE/ACM Transactions on*, 1(4):397–413, Aug 1993.

[4] Q. Huang, J. S. Smith, and T. Li. Web-based distributed embedded gateway system design. In *WI '06: Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 905–908, Washington, DC, USA, 2006. IEEE Computer Society.

[5] H. Jiang and C. Dovrolis. Why is the internet traffic bursty in short time scales? In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 241–252, New York, NY, USA, 2005. ACM Press.

[6] P. Koopman, J. Morris, and T. Maxino. Position paper: Deeply embedded survivability. *ARO Planning Workshop on Embedded Systems and Network Security, Raleigh NC*, Feb 22-23 2007.

[7] T. Lakshman, A. Neidhardt, and T. Ott. The drop from front strategy in TCP and in TCP over ATM. *INFOCOM '96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, 3:1242–1250 vol.3, Mar 1996.

[8] A. Mankin. Random drop congestion control. *SIGCOMM Comput. Commun. Rev.*, 20(4):1–7, 1990.

[9] J. Nilsson. *Real-Time Control Systems with Delays*. PhD thesis, Lund Institute of Technology, 1998.

[10] Robert Bosch, GmbH. CAN specification, version 2.0, 1991.

[11] J. Samuel. Emission related diagnostic services. *IEEE Communication Standards for European On-Board-Diagnostics Seminar (Ref. No. 1998/294)*, pages 10/1–10/6, Feb 1998.

[12] H. Sirisena, A. Haider, and K. Pawlikowski. Autotuning RED for accurate queue control. *Global Telecommunications Conference, 2002. GLOBECOM '02. IEEE*, 2:2010–2015 vol.2, Nov. 2002.

[13] TTA-Group. Time-triggered protocol TTP/C, high-level specification document, protocol version 1.1. Request online: `http://www.ttagroup.org/technology/specification.htm`, 2003.