Application of Congestion Notifications in a Real-Time Distributed System

Stephen Jackson and Dr. Bruce McMillin

Missouri University of Science & Technology, Rolla, MO 65409, USA,
{scj7t4,ff}@mst.edu

Abstract. This work presents a new technique for protecting a cyber-physical, real-time, distributed system during network congestion. New applications of existing congestion avoidance and detection methods are used to ensure that the system can still perform useful work, without causing physical instability, during network congestion. By notifying processes of congestion, a process adjusts its behavior to continue to effectively manage physical devices during network congestion. We demonstrate with these techniques that the availability of the distributed system is improved under various scenarios.

Keywords: computational geometry, graph theory, Hamilton cycles

1 Introduction

The **FREEDM!** (**FREEDM!**)[**?**] smart-grid is a project focused on the future of the electrical grid. This smart grid project is an advanced cyber-physical system; it couples distributed cyber controllers with controllable physical devices. Major proposed features of the **FREEDM! CPS!** (**CPS!**) rely on distributed local energy storage, and distributed local energy generation [**?**]. This vein of research emphasizes decentralizing the power grid, making it more reliable by distributing energy production devices. The **DGI!** (**DGI!**) is a critical component of this system. The **DGI!** performs automatic distributed configuration and management of power devices to manage an electrical grid. Energy management algorithms, a core feature of the **DGI!**[**?**], balance power in a smart-grid **CPS!**. These algorithms must have access to a set of available processes to work with. Automatic reconfiguration using a group management algorithm allows algorithms like [**?**][**?**][**?**] to autonomously control distributed power devices.

Because the **FREEDM!** smart-grid manages critical infrastructure in a distributed manner, its behavior during cyber or network fault conditions is of particular interest. In this work, we consider the effects of network congestion on a cyber communication network used by the **FREEDM!** smart-grid. We create a model version of a large number of **DGI!** processes in a simple partitioned setup. In the **FREEDM!** smart-grid, the consequences of this congestion could result in several problematic scenarios. First, if the congestion prevents the **DGI!** from autonomously configuring using its group management system, processes cannot work together to manage power devices. Secondly, if messages arrive too late, or are lost, the **DGI!** could apply settings to the attached power devices that drive the physical network to instability. These unstable settings could lead to problems in the power-grid like frequency instability, blackouts, and voltage collapse.

Additionally, this work has applications for other distributed systems. For example, in **VANET!** (**VANET!**)s[**?**][**?**], an attacker could overwhelm the ad-hoc network and prevent critical information about vehicle position and speed from being delivered in a timely manner. As a consequence, the affected vehicles could collide, endangering the passengers. Similarly, in a scenario with drone control or air-traffic control[**?**][**?**], congestion denies the delivery of important messages and could cripple that infrastructure.

In this work, we propose a technique to inform distributed processes of congestion. These processes act on this information to change their behavior in anticipation of message delays or loss. This behavior allows them to harden themselves against the congestion, and allows them to continue operating as normally as possible during the congestion. This technique involves changing the behavior of both the leader election[**?**] and load balancing algorithm during congestion.

To accomplish this, we extend existing networking concepts of **RED!** (**RED!**), **ECN!** (**ECN!**)[**?**], and ICMP source quench[**?**]. When a network device detects congestion, it notifies processes that the network is experiencing congestion and they should react appropriately. We propose a practical application for these techniques in scenarios where the message delay is a primary concern, not the rate at which data is sent. With this information, they can adjust their behavior to compensate for the detected congestion.

In this work we demonstrate an implementation of the **FREEDM! DGI!** in a **NS3!** (**NS3!**) simulation environment[**?**] with our congestion detection feature. The **DGI!** operates normally until the simulation introduces a traffic flow that congests the network devices in the simulation. After congestion has been identified by the **RED!** queueing algorithm, the **DGI!**s are informed. We show that when the congestion notifications are introduced, the **DGI!** maintains configurations which they would normally be unable to maintain during congestion. Additionally, we show a greater amount of work can be done without the work causing unstable power settings to be applied.

In Section **??**, we discuss technologies and techniques used in this work. Section **??** describes the implementation of the hardening technique and justifies the changes in terms of the message complexity and behavior of the system. Section **??** describes the experimental setup used to validate the techniques in this paper. Section **??** lists the results from our experiments. Finally, **??** summarizes our results.

2 Background

2.1 DGI

The DGI is composed of controllers each with a scheduler and a series of modules which implement various features. Features provided by the DGI modules include automatic configuration, power management, and state collection. The **DGI!** executes these modules using a round-robin real-time schedule. Processes synchronize their clocks and execute modules semi-synchronously. The load balancing module shares a quantum of power between a supply process and a demand process by performing a "migration." In a migration, power devices are manipulated to share power on a shared bus. Each time the load balancing module is scheduled to execute it performs multiple migrations during its execution phase. This schedule is depicted in Figure **??**. In the figure, each time the load balancing module runs it has the opportunity to complete a fixed number of migrations during its execution window. The schedule for the **DGI!** is decided before the process is started and does not change when the **DGI!** is running. All **DGI!** processes that can potentially group together use the same schedule.

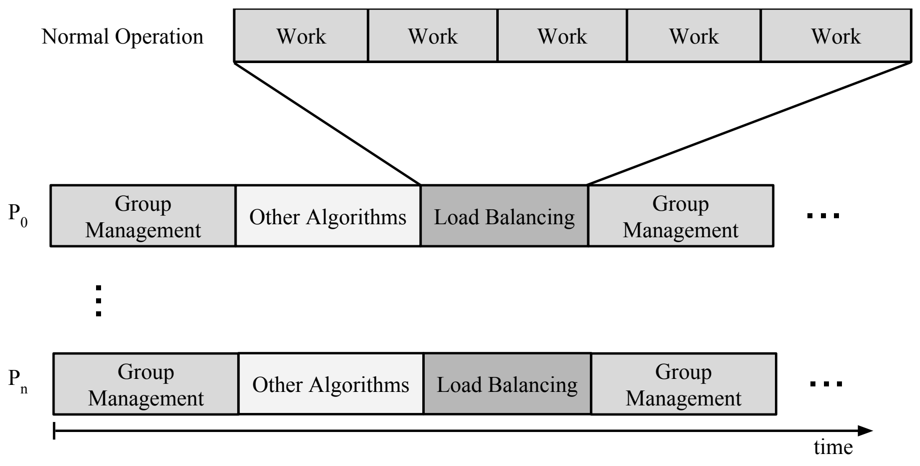

Fig. 1. Example of a **DGI!** schedule. During the execution of the Load Balancing module, work is performed to manage power devices.

Leader Election The **DGI!** uses a variation on the leader election algorithm, "Invitation Election Algorithm," written by Garcia-Molina[**?**]. This algorithm provides a robust election procedure which allows for transient partitions. Transient partitions are formed when a faulty link inside a group of processes causes the group to divide temporarily. These transient partitions merge when the link becomes more reliable.

The elected leader is responsible for making work assignments, identifying and merging with other coordinators when they are found, and maintaining an up-to-date list of peers. Group members monitor the group leader by periodically checking if the group leader is still alive by sending a message. If the leader fails to respond, the querying peers will enter a recovery state and operate alone until they can identify another coordinator. Therefore, a leader and each of the members maintain a set of currently reachable processes, a subset of all known processes in the system.

Using a leader election algorithm allows the **FREEDM!** system to autonomously reconfigure rapidly in the event of a failure. Cyber components are tightly coupled with the physical components, and reaction to faults is not limited to faults originating in the cyber domain. Processes automatically react to crash-stop failures, network issues, and power system faults. The automatic reconfiguration allows processes to react immediately to issues, faster than a human operator, without relying on a central configuration point. However, it is important the configuration a leader election supplies is one where the system can do viable work without causing physical faults like voltage collapse or blackouts[**?**].

Processes determine the optimal coordinator (the one with the lowest process ID) through the exchange of **AYT!** (**AYT!**) and **AYC!** (**AYC!**) messages. A process will only accept invites from processes more optimal than itself, and more optimal than its current leader. Once a timeout expires, the coordinator will send a "Ready" message with a list of peers to all processes that accepted the invite. To prevent live-lock the processes do not leave their previous group until the new Ready message arrives, unlike the original "Invitation Election Algorithm." The invited processes have timeouts for when they expect the ready message to arrive.

Once a group is formed it must be maintained. To do this, processes occasionally exchange messages to verify the other is still reachable. Coordinators send **AYC!** messages to members of its group to check if the process has left the group. Group members send **AYT!** messages to the coordinator to verify they haven't been removed from the group, and to ensure the coordinator is still alive. If processes fail to reply to received message before a timeout, they will leave the group. Leaving the group can either be caused by the coordinator removing the process, or the process can enter a recovery state and leave the group, forming a new group by itself.

This modified algorithm allows the interactions of the processes in our semi-synchronous system to be modeled with a Markov chain. A version of this algorithm with a restriction on what processes could become coordinator is presented in [**?**]. We elected to use an improved version of the algorithm in this work because of its easy-to-follow group state.

Power Management In this work we utilize the load balancing algorithm from [**?**]. The load balancing algorithm performs work by managing power devices with a sequence of migrations[**?**]. In each migration, a sequence of message exchanges identify processes whose power devices are not sufficient to meet their local demand and other processes supply them with power by utilizing a shared bus. To do this, first processes that cannot meet their demand announce their need to all other processes. Processes with devices that exceed their demand offer their power to processes that announced their need. These processes perform a three-way handshake. At the end of the handshake, the two processes have issued commands to their attached devices to supply power from the shared bus and to draw power from the shared bus.

The **DGI!** algorithms can tolerate packet loss and is implemented using UDP to pass messages between **DGI!** processes. Effects of packet loss on the **DGI!**'s group management module have been explored in [**?**] and [**?**]. The load balancing algorithm can tolerate some message loss, but lost messages can cause migrations to only partially complete, which can cause instability in the physical network. A failed migration is diagrammed in Figures **??** and **??**.

With this power migration algorithm, uncompensated actions may occur in the power system. These actions can eventually lead to power instability through issues such as voltage collapse. Additionally, the supply process may not always be certain if the second half of the action was completed or not. If the "Draft Accept" message does

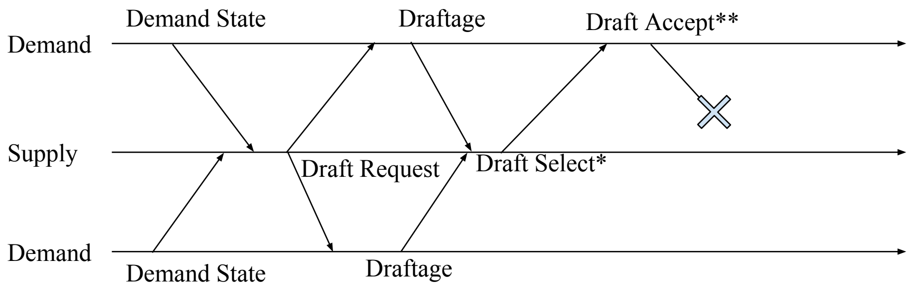

Fig. 2. Example of a failed migration. (*) and (**) mark moments when power devices change state to complete the physical component of the migration. In this scenario, the message confirming the demand side made the physical is lost, leaving the supply node uncertain.

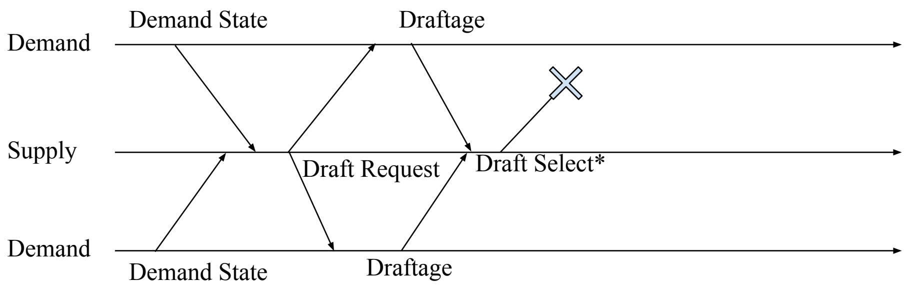

Fig. 3. Example of a failed migration. (*) marks a moment when power devices change state to complete the physical component of the migration. In this scenario, the supply process changes its device state, but the demand process does not.

not arrive from the demand process, the supply process cannot be certain of whether or not its "Draft Select" message was received. If the supply process takes action to compensate by reversing the migration and the confirmation arrives later the system will also be driven towards instability because another process completed an uncompensated action. Processes could potentially confirm the number of failed migrations with a state collection technique.

It is therefore desirable to manage the processes to minimize the number failed migrations. In this work, we consider effects due to delays by congestion.

2.2 Random Early Detection

The **RED!** queueing algorithm is a popular queueing algorithm for network devices. It uses a probabilistic model and an **EWMA!** (**EWMA!**) to determine if the average queue size exceeds predefined values. These values are used to identify potential congestion and manage it. This is accomplished by determining the average size of the queue, and then probabilistically dropping packets to maintain the size of the queue. In **RED!**, when the average queue size avg exceeds a minimum threshold (min_{th}), but is less than a maximum threshold (max_{th}), new packets arriving at the queue may be "marked". The probability a packet is marked is based on the following relation between p_b and p_a where p_a is the final probability a packet will be marked.

$$p_b = max_p(avg - min_{th})/(max_{th} - min_{th}) \tag{1}$$

$$p_a = p_b/(1 - count * p_b) \tag{2}$$

Where max_p is the maximum probability a packet will be marked when the queue size is between min_{th} and max_{th} and $count$ is the number of packets since the last marked packet. With this approach, p_b varies linearly with the average queue size, and the p_a is a function of p_b and the time since the last packet was marked. If avg is greater than max_{th}, the probability of marking increases from max_p to 1 as the average queue size approaches $2 * max_{th}$ In the event the queue fills completely, the **RED!** queue operates as a drop-tail queue.

In a simple implementation of the **RED!** algorithm, marked packets are dropped. For a TCP application, the result of the dropped packets causes the slow-start congestion control strategy to reduce the rate packets are sent. A more advanced implementation, using **ECN!**, sets specific bits in the TCP header to indicate congestion. By using **ECN!**, TCP connections can reduce their transmission rate without re-transmitting packets.

UDP applications have not typically utilized **ECN!**. Although the **ECN!** standard has flags in the IPv4 header, access to the IPv4 header is not possible on most systems, most notably Linux. Furthermore, there is not a "one size fits all" solution to congestion in UDP algorithms. However, for the **DGI!** and a class of similar real-time processes, congestion notification has great potential. If processes can adjust the amount of traffic they send based on the anticipated congestion (by disabling features, for example), they can decrease the effects of congestion.

3 Application

3.1 Usage Theory

When the **RED!** algorithm identifies congestion it must notify senders of congestion. Since the **ECN!** fields in IPv4 are not available to applications running on the system, the notifications are multicast onto the source interface. Additionally, since this approach is non-standard and most UDP applications would not understand the notification, we have opted to create an application that runs on the network device. This application is responsible for generating the multicast message. It also keeps a register of hosts running applications that support reacting to the **ECN!** notification.

When the **RED!** algorithm detects congestion, it sends a multicast beacon to a group of interfaces informing of the level of congestion. For similarity with the **RED!** algorithm and the **NS3!** implementation, this notification is classified as either “soft” or “hard.” A soft notification is an indication the congestion in the network is approaching a level where real-time processes can expect message delays that may affect their normal operation. A hard notification indicates the congestion has reached a level where messages may be subject to both delay and loss. The notifications are rate limited so they do not flood the network.

3.2 Group Management

The group management module’s execution schedule is broken into several periods of message generation and response windows. Because the schedule of the **DGI!** triggers the execution of group management modules approximately simultaneously, the traffic generated by modules is bursty. The number of messages sent is $O(n^2)$ (where n is the number of processes in the system), in a brief window, which is dependent on how well the clocks are synchronized in the system. The duration of the response window is dependent on the amount of time it takes for messages to propagate to the hardest-to-reach process the **DGI!** hopes to group with. Additionally, to contend with congestion, an additional slack must be added to allow the **RED!** algorithm to detect congestion before it reaches a critical level.

Figure **??** depicts typical queueing behavior for a network device serving **DGI!** processes under different circumstances. Because the traffic generated by **DGI!** modules is very bursty, the queue experiences a phenomena where the bursty traffic mixed with a steady background traffic causes the queue to fill. With no background traffic, the impulse queues a large number of messages, but those messages are distributed in a timely manner. When the background traffic is introduced, the queue takes longer to empty. At a critical threshold, the queue does not empty completely before the next burst is generated by the **DGI!**. In this scenario, the queue completely fills and no messages can be distributed. The **RED!** algorithm and **ECN!** are used to delay or prevent the queue from reaching this critical threshold.

For this work, the algorithm from [**?**] was used. This algorithm has a higher message complexity when in a group than the Garcia-Molina algorithm it is based on. However, it does possess a desirable memoryless property that makes it easy to analyze. This work uses an improved version of the algorithm which removes the restrictions in [**?**] where only one process could become the leader.

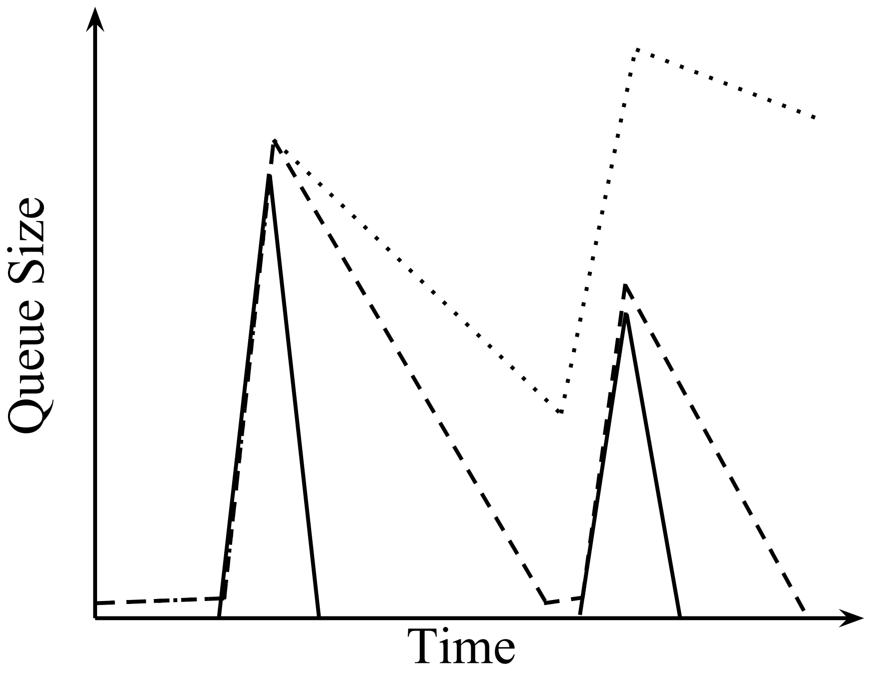

Fig. 4. Example of network queueing during **DGI!** operation. **DGI!** modules are semi-synchronous, and create bursty traffic on the network. When there is no other traffic on the network (solid line), the bursty traffic causes a large number of packets to queue quickly, but the queue empties at a similar rate. With background traffic (dashed line), the bursty traffic causes a large number of packets to be queued suddenly. More packets arrive continuously, causing the queue to drain off more slowly. When the background traffic reaches a certain threshold (dotted line), the queue does not empty before the next burst occurs. When this happens, messages will not be delivered in time, and the queue will completely fill.

Soft ECN! A soft **ECN!** message indicates the network has reached a level of congestion where the router suspects processes will not be able to meet their real time requirements. The soft **ECN!** message encourages the **DGI!** processes to reduce the number of messages they send to reduce the amount of congestion they contribute to the network, and to allow for reliable distribution techniques to have additional time to deliver messages (since fewer messages are being sent). In the case of potential congestion, the group management module can reduce its traffic bursts by disabling elections during the congestion. When the elections are disabled, messages for group management are only sent to members of the group. Processes do not seek out better or other leaders to merge with. As a consequence, the message complexity for processes responding to the congestion notification reduces from $O(n^2)$ to $O(n)$.

Hard ECN! In a hard **ECN!** scenario, the router will have determined congestion has reached a threshold where the real-time processes will soon not be able to meet their deadlines. In this scenario, the real-time process will likely split its group. In an uncontrolled situation, the split will be random. It is therefore desirable when this level of traffic is reached to split the group. Splitting the group reduces the number of messages sent across the router for modules with $O(n^2)$ (where n is the number of processes in the original group) message complexity. For larger groups, splitting them provides a significant savings in the number of messages that must be queued by the router, especially since the traffic is very bursty.

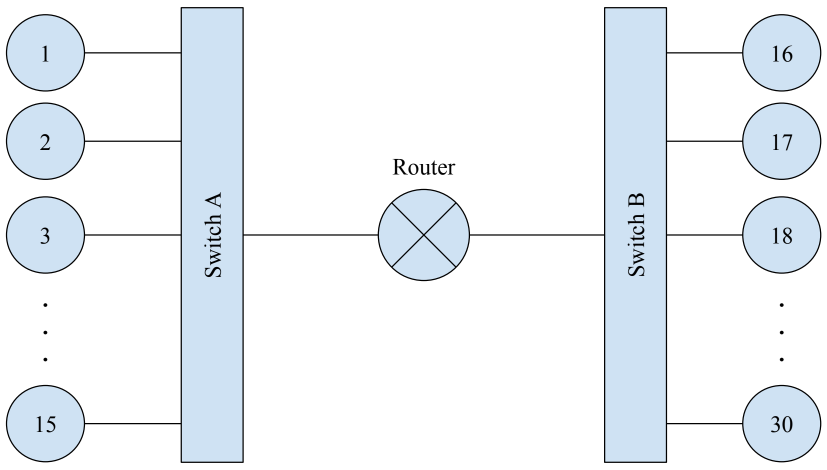

Fig. 5. Example of process organization used in this paper. Two groups of processes are connected by a router.

Suppose a network like one depicted in Figure **??**, where processes are divided by a router. In Figure **??**, there are n processes on one side of the network and m on the other. In normal operation the omission-modelable algorithm has an $O(n^2)$ message complexity. In Soft **ECN!** maintenance mode, the reduced number of messages reduces the complexity to $O(n)$ by disabling elections.

During elections (and with each group update) the leader distributes a fallback configuration that will coordinate the division of the groups during intense congestion. When the **ECN!** notification is received the processes will halt all current group management operations and enter a splitting mode where they switch to the fallback configuration. The leader of the group distributes a fallback notification to ensure all processes in the group apply their new configuration. The complexity of distributing the notification is linear $O(n)$ and processes that already received the notification will have halted their communication. This approach will ideally avoid the burst/drain phenomena from figure **??**.

The design of the fallback configuration can be created to optimize various factors. These factors include cyber considerations, such as the likely network path the processes in the group will use to communicate. By selecting the group around the network resources, the group can be selected to minimize the amount of traffic that crosses the congested links in the future. Additionally, considerations from the physical network can be considered. Fallback groups can be created to ensure they can continue to facilitate the needs of the members. This can take into the consideration the distribution of supply and demand processes in the current group. By having a good mix of process types in the fallback group the potential for work can remain high.

3.3 Cyber-Physical System

For a real-time **CPS!**, message delays could affect coordinated actions. As result, these actions may not happen at the correct moments or at all. Since the two-army problem prevents any process from being entirely certain a coordinated action will happen in concert, problems arising from delay or omission of messages is of particular interest. In particular, we are interested in the scenario from [**?**], where only half of a power migration is performed. Other power management algorithms could have similar effects on the power system based on this idea of a process performing an action that is not compensated for by other processes.

Soft ECN! In a soft congestion notification mode, the process being informed of the congestion can reduce its affect on the congestion by changing how often it generates bursty traffic. Processes running the load balancing algorithm make several traffic bursts when they exchange state information and prepare migrations. As shown before, if the interval between these bursts is not sufficient for the queue to drain before the next burst occurs, then critical, overwhelming congestion occurs. Since the schedule of the **DGI!** is fixed at run-time processes cannot simply extend the duration of the load balancing execution phase. However, on notification from the leader, the process can, instead, adjust the number of migrations to increase the message delivery interval. This notification to reduce the schedule originates from the coordinator as part of the message exchange necessary for the process to remain in the group. Every process in the group must receive this message to participate in load balancing, ensuring all processes remain on the same real-time schedule. Using this approach, the amount of traffic generated is unchanged but the time period a process waits for the messages to be distributed is increased.

Hard ECN! When the **DGI!** process receives a hard congestion notification, the processes switch to a predetermined fallback configuration. This configuration creates a cyber partition. By partitioning the network, the number of messages sent by applications with $O(n^2)$ message complexity can be reduced significantly. Each migration of load balancing algorithm begins with an $O(n^2)$ message burst and so benefits from the reduced group size created by the partition.

Suppose there is a network like the in Figure **??** with n processes on one half and m on the other. The number of messages sent across the router for the undivided group is of the order $2mn$ as the n processes on side A send a message to the m on side B and vice-versa. Let i_1 and j_1 be the number of processes from side A and side B (respectively) in the first group created by the partition. Let i_2 and j_2 be the number of processes in the second group created by the partition under the same circumstances of i_1 and j_1. The number of messages sent that pass through the router, is then

$$2i_1j_1 + 2i_2j_2 \tag{3}$$

For an arbitrary group division, the following can be observed. Suppose i_1 and j_2 are the cardinality of two arbitrarily chosen sets of processes from side A and side B respectively. Following the same cut requirements as before:

$$i_2 = n - i_1 \tag{4}$$

$$j_2 = m - j_1 \tag{5}$$

The the number of messages that must pass through the router for this cut is:

$$2i_1j_1 + 2(n - i_1)(m - j_1) \tag{6}$$

$$= 2i_1j_1 + 2(nm - mi_1 - nj_1 + i_1j_1) \tag{7}$$

$$= 2(2i_1j_1 + nm - mi_1 - nj_1) \tag{8}$$

The value is maximized when i_1 and j_1 are $\frac{n}{2}$ and $\frac{m}{2}$:

$$2(2\frac{mn}{4} + mn - \frac{mn}{2} - \frac{mn}{2}) \tag{9}$$

$$= mn \tag{10}$$

Which is a reduction of half as many messages. For systems with a large number of participating processes this represents a significant reduction in the number of messages sent across the router. As a consequence, this further extends the delivery window for processes sending messages.

In the best case scenario, some cut will have a single process opposite a large number of processes. Consider cut where one process is selected from side A and $m - 1$ are selected from side B. The cut will also create a second group of $n-1$ processes from side A and one process from side B.

$$2(m-1)+2(n-1) \tag{11}$$

Which represents a reduction in message complexity from the original $2mn$.

3.4 Relation To Omission Model

The synchronization of clocks in the environment is assumed to be normally distributed around a true time value provided by the simulation. The shape of the curve created by plotting the queue resembles that of the **CDF!** (**CDF!**) of the normal distribution, noted $F(x)$. A simple description of the traffic behavior can then be described in terms of that curve. First, observe that when the queue hits a specific threshold, even if the queue is drained at an optimal rate, the nth queued packed will not be delivered in time:

$$Qsize - min(Qsize, (DequeueRate * \Delta t)) \geq 0 \tag{12}$$

Where Δt is the deadline for the message to be delivered. If the size of the queue exceeds the number of messages that can be delivered before Δt passes, some messages will not be delivered. The size of the queue during the message bursts created by the DGI depends on the message complexity of the algorithm, the number of messages already in the queue, the other traffic on the network, and any replies that also have to be delivered in that interval. Therefore, let c represent the rate that traffic is generated by other processes. Let $init_q$ represent the number of messages in the queue at the start of a burst. Let $init_m$ represent the number of messages sent in the beginning of the burst. Let $resp$ represent the number of messages sent in response to the burst that must still be delivered before Δt passes. We can then express $Qsize$ as two parts:

$$Qsize = Burst + Obligations \tag{13}$$

Where $Burst$ takes the form of the **CDF!** for the normal distribution:

$$Burst = init_m * F(x) \tag{14}$$

$$Obligations = c * \Delta t + init_q + resp \tag{15}$$

From this we can derive the equation:

$$F(x) \geq \frac{DequeueRate * \Delta t - c * \Delta t - init_q - resp}{init_m} \tag{16}$$

Where, from Equation **??**, $DequeueRate * \Delta t$ is less than or equal to the number of messages in the queue. Solving for $F(x)$ gives a worst case estimate of the omission rate for a specific algorithmic or network circumstance. $DequeueRate$ is affected by the amount of traffic in the system. It should be obvious a greater amount of background traffic corresponds to a greater average queue size. From an relationship between the background traffic and the average queue size and the results presented in [**?**], Equation **??** can be used to select the ECN parameters.

4 Experimental Setup

Experiments were run in a Network Simulator 3.23[**?**] test environment. The simulation time replaced the wall clock time in the **DGI!** for the purpose of triggering real-time events. As a result, the computation time on the **DGI!**s for processing and preparing messages was neglected. However, to compensate for the lack of processing time, the synchronization of the **DGI!**s was instead randomly distributed as a normal distribution. This was done to introduce realism to ensure events did not occur simultaneously. Additionally, the real-time schedules used by the **DGI!** were adjusted to remove the processing time that was neglected in the simulation.

The **DGI!**s were placed into a partitioned environment. The test included 30 nodes. Each of the nodes ran one **DGI!** process. Two sets of 15 **DGI!** were each connect to a switch and each switch was in turn connected to the router. This network is pictured in Figure **??**. Node identifiers were randomly assigned to nodes in the simulation and used as the process identifier for the **DGI!**.

The links between the router and the switches had a **RED!** enabled queue placed on both network interfaces. The **RED!** parameters for all queues were set identically. A summary of **RED!** parameters are listed in Table **??**. All links in the simulation were 100Mbps links with a 0.5ms delay. RED was used in packet count mode to determine congestion. ARP tables were populated before the simulation began.

RED! parameters were selected using Equation **??** and [**?**]. The relationship between the background traffic and the average queue size was estimated through runs of the **NS3!** simulation. Figure **??** demonstrates the observed relationship between the total background traffic and the maximum average queue size for that level of traffic. Additionally, the $DequeueRate$ was collected from a run of the simulation without traffic, and was found to be 713.08 packets/second. Therefore, from Equation **??**, assuming $init_q = 0, resp = 225, init_m = 225$ and $\Delta t = 1$, the maximum traffic rate with no omissions is 263.0 packets/second. The number of packets for the $resp$ and $init_m$ were selected from the worst case of the algorithm in [**?**]. Based on the traffic parameters in Table **??**, 263.0 packets/second corresponds to 1.077 Mbps of traffic generated at one switch and 2.1545 Mbps traffic overall. From the polynomial estimate in Figure **??**, the maximum average queue size for that level of traffic is 94.715, estimated as 90 for the **RED!** Min Threshold in Table **??**. RED Max Threshold is computed using a similar technique, but using the message complexity for the Load Balancing algorithm, since it maintains its complexity during Soft ECN mode.

To introduce traffic, processes attached to each of the switches attempted to send a high volume of messages to each other across the router. The number of packets sent per second was a function of the data rate and the size of the packets sent. In each simulation, half of the traffic originated from each switch. Due to the bottleneck due to the properties of the network links, the greatest queueing effect occurred at the switches.

5 Results

Figures **??**, **??** and **??** show the normal operation of the system. In this configuration, there is no congestion on the network. The **DGI!**s start, group together and then begin

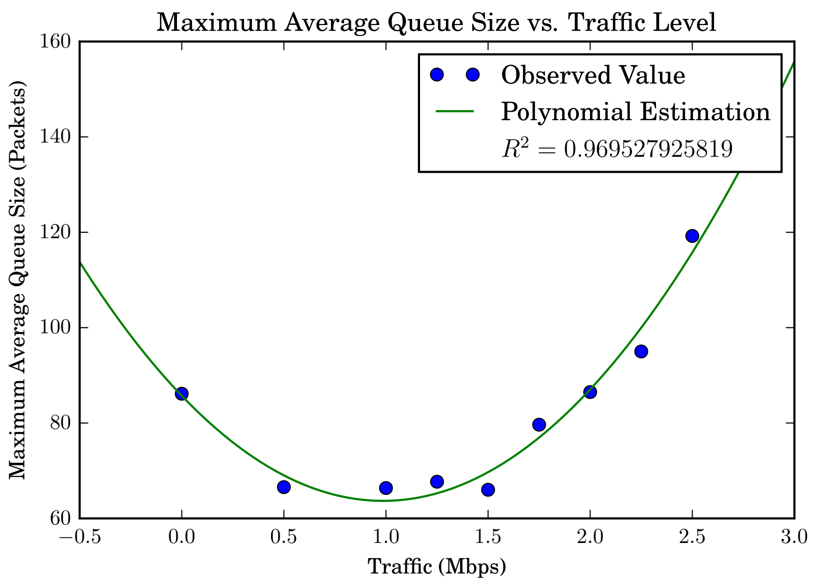

Fig. 6. Plot of the maximum observed average queue size as a function of the overall background traffic. The polynomial estimate is $y = 22.70x^2 - 44.74x + 85.72$

Parameter	Value
RED Queueing Mode	Packet
RED Gentle Mode	True
RED Q_w	0.002
RED Wait Mode	True
RED Min Threshold	90
RED Max Threshold	130
Maximum Queue Size	1000
RED Link Speed	100 Mbps
RED Link Delay	0.5 ms
Clock Distribution σ	0.005
Traffic Packet Size	512 Bytes

Table 1. Summary of **RED!** parameters. Unspecified values default to the **NS3!** implementation default value

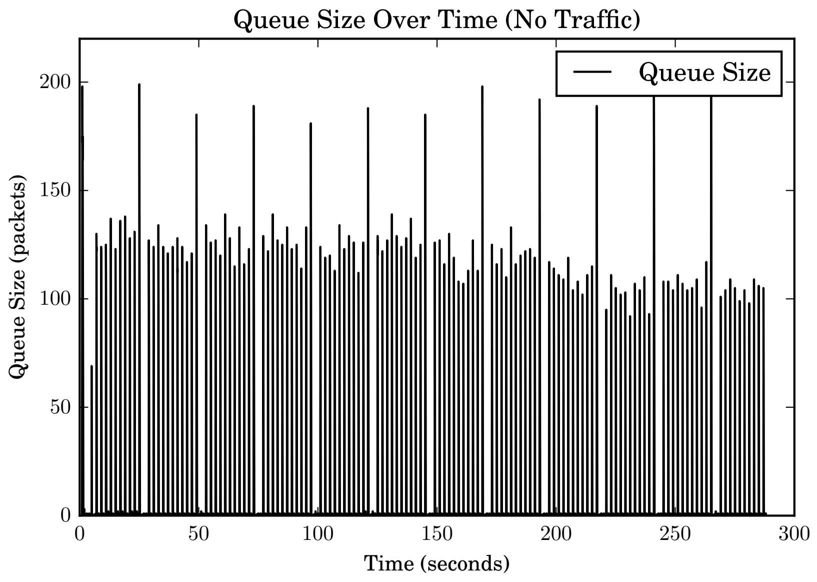

Fig. 7. Plot of the queue size for a queue used to send traffic from switch A to the router. In this scenario, the only traffic is created by the **DGI!**. The largest peaks are created by the execution of the Group Management module. Load Balancing migrations produce each of the smaller peaks.

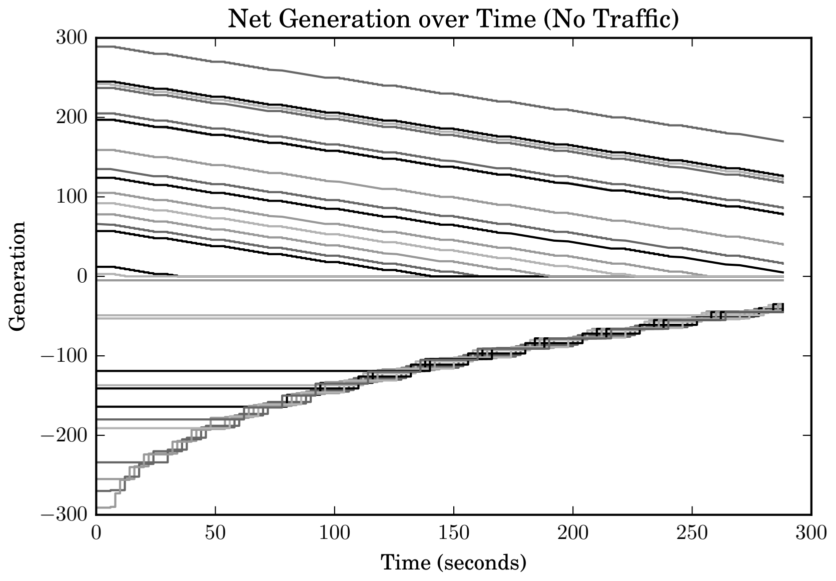

Fig. 8. Plot of the net generation for each process in the system. As processes migrate power, their net generation moves towards zero.

migrating power between processes. Figure **??** plots the queue size over time for a queue used to send packets from a switch to the router. Figure **??** is a detailed view of a portion of Figure **??**. Figure **??** shows the queue size during the normal operation of group management as well as the first migration of the load balancing module. The dotted line plots the **EWMA!** of the size of the queue. From this experiment we establish the min_{th} value used as a **RED!** queue parameter. The traffic generated by each step of the group management algorithm is very bursty. It should be obvious that the tightness of the clock synchronization in the group affect how large this peak is. Figure **??** plots the power level of the processes in the system. Like [**?**], the level of the power at a process is the net sum of its power generation capability and load. As power is shared on the network, processes with excess generation, converge toward zero net power. Demand processes also converge toward zero net power. The number of migrations completed is a good metric for how much work can be performed.

Figure **??** shows the queue size as the network traffic begins to increase. The **DGI!**s in these experiments use a schedule that allows for some congestion to occur before processes are disrupted. This slack gives the network devices the opportunity to identify when the network congestion will go beyond the acceptable threshold.

Figure **??** shows an example of congestion affecting the physical network without **ECN!**. As a result of the congestion in Figure **??**, processes leave the main group. Additionally power migrations are affected: migrations are lost, or the supply process is

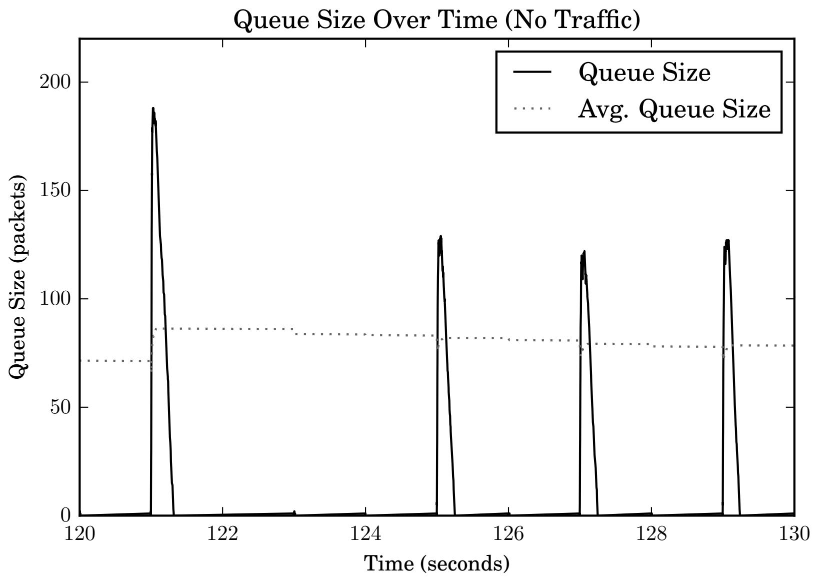

Fig. 9. Detailed view of Figure **??**. The left most peak is from the maintenance of a group, and the 3 smaller peaks are from Load Balancing migrations. The average queue size is only updated with a new packet is enqueued.

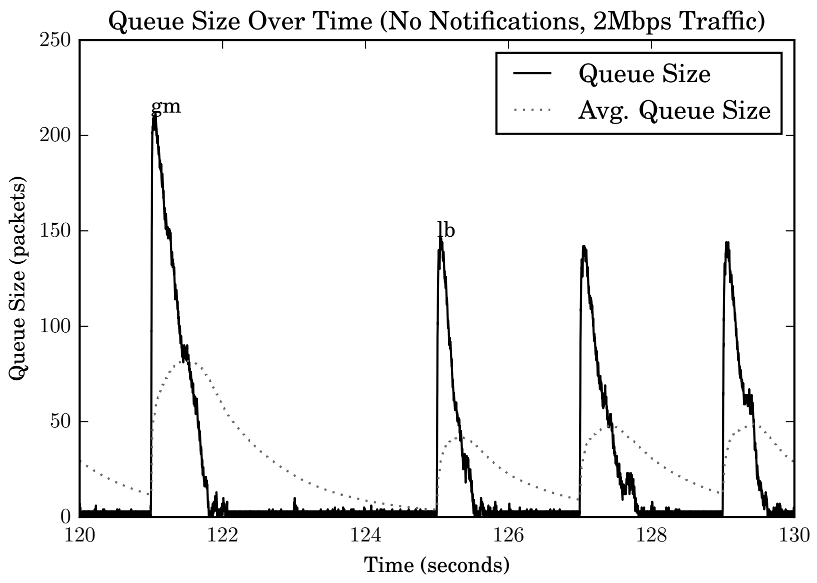

Fig. 10. Detailed view of the effect on queue size as other network traffic is introduced. Compared to Figure **??**, the peaks are taller and wider. Background traffic causes the average queue size to be updated more frequently, however, the average queue size is not significantly affected by the added traffic.

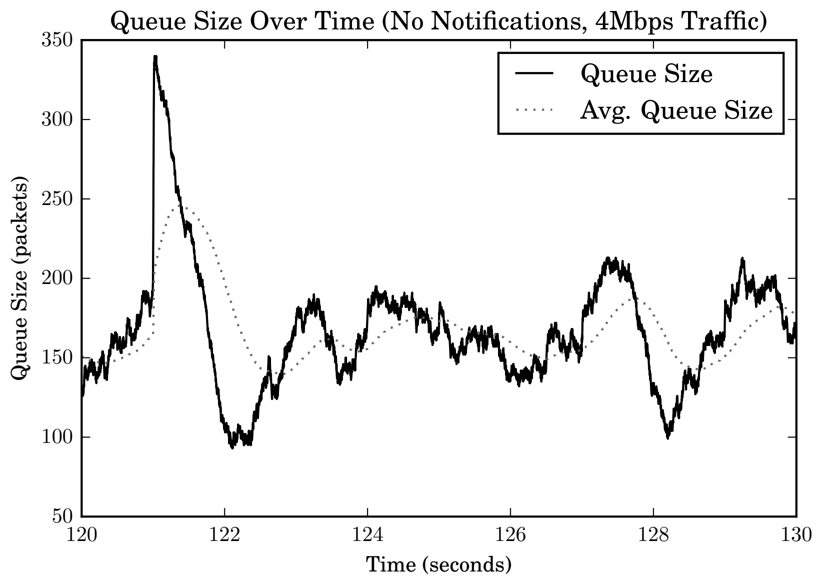

Fig. 11. Detailed view of the effect on queue size as other network traffic is introduced. With no **ECN!** notifications, the peak from Group Management is much larger. The congestion is sufficient that at this moment in the simulation, 9 processes have left the primary group, leading to the smaller load balancing peaks.

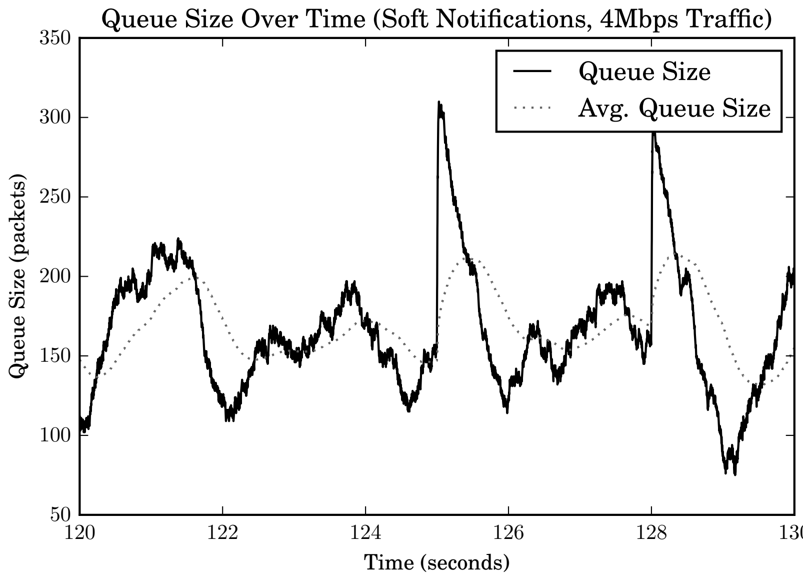

Fig. 12. Detailed view of the effect on queue size as other network traffic is introduced. In this scenario, the ECN notifications put Group Management into a maintenance mode that reduces its message complexity and switches Load Balancing to slower migration schedule. As a result, the average queue size stay closer to 200 packets, the primary group does not split, and no migrations are lost.

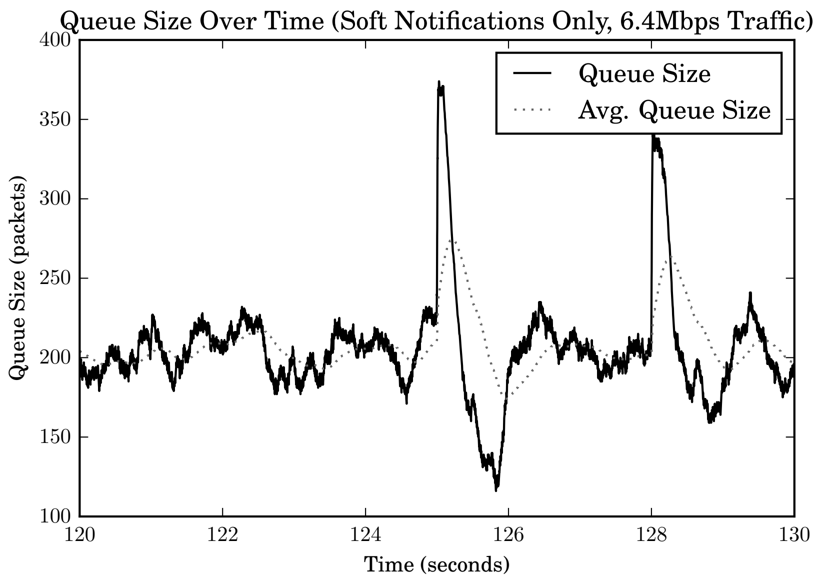

Fig. 13. Detailed view of the effect on queue size as a large amount network traffic is introduced. Groups are unstable and processes occasionally leave the main group. The group management peak is small because of the soft notification and load balancing peaks are large because the main group is still most of the processes (28 of 30 at the plotted interval). Several migrations are lost due to delays.

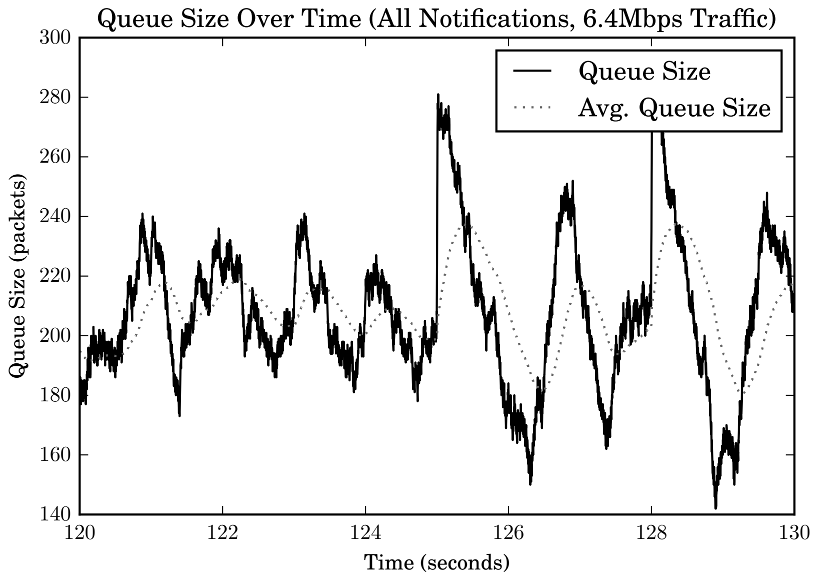

Fig. 14. Detailed view of the effect on queue size as a large amount of network traffic is introduced. In this scenario, hard notifications cause the groups to divide. As a result of the smaller groups, the group management and load balancing peaks are smaller than those in **??**. No migrations are lost during execution.

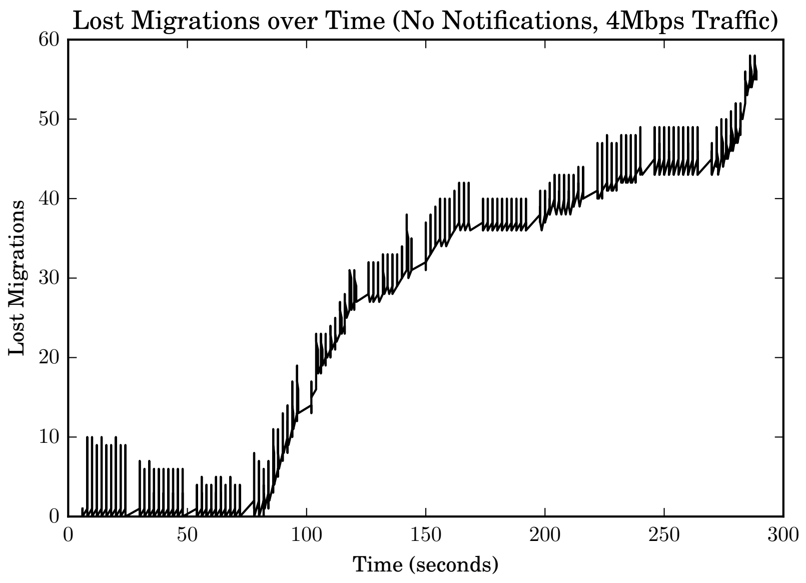

Fig. 15. Count of lost migrations from all processes over time. As migrations proceed, some migrations are counted as lost until the second process confirms it has been completed. In this scenario without congestion management, a large number of migrations are lost as a result of message delay.

left uncertain of migrations completions. Figure **??** plots the count of failed migrations over time.

Figure **??** shows an example of the **ECN!** algorithm notifying processes of the congestion. Compared to the scenario in Figure **??**, the **ECN!** algorithm successfully prevents the group from dividing, and increases the number of migrations by reducing the number of attempted migrations each round.

Figures **??** and **??** show an example of a more extreme congestion scenario. In Figure **??**, the **RED!** algorithm shares a Hard **ECN!** notification. This notification causes the **DGI!** to switch to a smaller fallback configuration. This fallback configuration decreases the queue usage from Figure **??** to Figure **??**. Without this fallback configuration behavior, the system is greatly affected by the traffic.

6 Conclusion

In this work, we presented a technique for hardening a real-time distributed cyber-physical system against network congestion. The **RED!** queueing algorithm and an out-of-band version of explicit congestion notification (ECN) were used to signal an application of congestion. Using this technique the application changed several of its characteristics to ready itself for the increased message delays caused by the congestion.

These techniques were demonstrated on the **DGI!**, a distributed control system for the **FREEDM!** smart-grid project. In particular, this paper demonstrated the hardening techniques were effective in keeping the **DGI!** processes grouped together. Additionally, it helped ensure the changes applied to the **DGI!** through cyber-coordinated actions did not destabilize the physical power network.

This technique will be important to create a robust, reliable **CPS!** for managing future smart-grids. However, this technique could potentially be applied to any **CPS!** that could experience congestion on its network, as long as it has the flexibility to change its operating mode. Potential applications can apply to both the cyber control network and the physically controlled process. For example, in a **VANET!** system, the vehicles could react to congestion by increasing their following distance.

References