Argus: Bandwidth Management in Sensor Networks

Cheng Tien Ee
Department of Computer Science
University of California, Berkeley

***Abstract*— Wireless, multi-hoping sensor networks with more than tens of motes often encounter traffic congestion. The occurrence of congestion subsequently results in loss of control over the bandwidth each mote effectively receives. In this paper we present a bandwidth management mechanism for sensor networks called *Argus* that incorporates congestion control and the ability to divide available bandwidth amongst traffic flows and as well as handle different traffic patterns. Argus focuses primarily on the transfer of data packets, and is implemented in the routing and transport layers. Since it independent of the data-link layer, it can therefore be applied to a network with different MAC protocols. Argus implements Extended Epoch-based Proportional Selection (EEPS), a distributed variant of Weighted Fair Queuing. EEPS performs scheduling of per-child queues, with the associated weights primarily derived from the flow sizes of downstream motes. Since a mote only requires knowledge of each child mote's total flow and not each individual flow's size, and since queues need be maintained on a per-child rather than a per-flow basis, Argus is scalable and is ideal for resource-constrained sensor networks. We demonstrate that Argus is able to achieve its goals of bandwidth management using a 26-mote mica2dot testbed.**

Key words: sensor networks, distributed algorithms, weighted fair queuing, congestion control, bandwidth management

I. Introduction

Since the first sensor mote was created, a multitude of networking protocols have been designed, written and tested. These protocols have differed greatly from the traditional networking protocols found in the Internet. For instance, unlike the traditional Internet, where dedicated routers perform the highly specialized task of routing packets from one network to another, each sensor mote is capable of generating data as well as forwarding data upstream towards a sink mote, or *base station*. Also, bandwidth management in the Internet is typically accomplished at two different entities: gateway routers and end-hosts. Such management in sensor networks will have to be for traffic generated locally as well as route through traffic. Yet another difference is the homogeneity of the networks. Thus far, sensor networks have been mostly homogeneous, in that the type of motes, their capabilities and resources, are similar if not the same. Sensor networks are also homogeneous in the traffic management sense: each packet in a sensor network is treated the same as all other packets, there is no concept of data flows from each source mote to the corresponding sink mote. Although such a scheme is easy to implement and consumes little state in the motes, it results in a loss of control in the management of bandwidth. For instance, it is not possible to devote more bandwidth to a particular data flow, neither is it possible to ensure that the base station receives approximately the same number of packets from all motes. Also, different types of data may be simultaneously generated by the network: some data packets may have to be sent as is to the base station, whereas others may have contents that can be aggregated or processed in the network itself.

This loss of control over bandwidth distribution may not be significant if we consider simple one-hop networks of less than 10 motes, generating data at a lower rate than the maximum the network can handle. However, as the size of the network increases to tens or hundreds of motes, congestion occurs, causing a general increase in latency and dropping of packets and thus wastage of energy. Moreover, such networks typically cover a larger area and are thus more likely to be multi-hoping, which results in more packets from motes closer to the base station to be received. This phenomenon is observed even when congestion control and Automatic Repeat Request (ARQ) are implemented [8].

Increasing the size of the network to more than hundreds of motes has been widely acknowledged to require a change in architectural design. Rather than having a thousand motes send data to one base station, it is preferable to partition the network into distinct regions, each with its own base station. We can extend this scenario to the one shown in Figure 1. Here, the infrastructure motes route data from the patch of type A sensor motes as well as the type B sensor motes to the gateway device, which in turn transmits the data over the Internet to a location where they can be further processed.

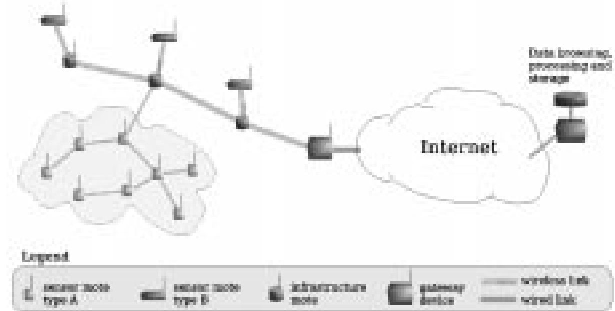

Fig. 1. A heterogeneous sensor network, comprising of motes with different resources. Data is gathered and may be processed in-network before being sent to a center through the Internet.

Such an architectural design serves to increase the effective available bandwidth in the network, as well as to accommodate motes with different capabilities. It is reasonable to assume that infrastructure motes can have multiple Media Access Control (MAC) interfaces, for instance, they may have different radios for interacting with micaz and mica2dot motes. This is a different definition of heterogeneity that is not captured by

any high level sensor network protocol.

The traffic traversing a heterogeneous network can be potentially diverse in terms of the average rate and pattern. A mote may generate more packets when an event is detected, or the application running on the motes may aggregate received data or forward them unaltered. It is therefore difficult, if not impossible, to determine a good upper bound for the total amount of traffic to be carried at any one time. Together with the fact that quality of wireless links can fluctuate greatly, it is not feasible to exactly provision the network so that bandwidth is always available. Over-provisioning the network appears to be an attractive solution, but has the disadvantage of incurring additional cost and is not a solution if there are applications that attempt to transmit data as fast as they can. Over-provisioning is also not as simple as adding more motes to the network, since, unlike in the case of wired networks, having more wireless motes implies that in general interference increases thereby reducing the effective bandwidth. Furthermore, different applications are likely to have varying bandwidth requirements relative to one another. Since it is intractable for an application programmer to be aware of all possible applications running on the same network and adjust his bandwidth usage accordingly, we believe that having a common, automatically and dynamically self-adjusting bandwidth management mechanism amongst all sensor motes is a better solution.

We propose an Extended Epoch-Based Proportional Selection (EEPS) algorithm that is a distributed variant of Weighted Fair Queuing (WFQ) [2], where the weights involved are derived from the flow sizes of downstream motes and may be adjusted by other factors. We call our implementation *Argus*; it requires per-child queues and only maintains state pertaining to its one-hop neighboring motes. Per-flow management is required only at the originating mote, and not at intermediate motes. Communication between upstream and downstream motes is minimized and is piggybacked onto data packets, thus eliminating the use of explicit control traffic in the network.

The rest of this paper is organized as follows: Section II briefly covers related work in this area; Section III states the goals of our bandwidth management mechanism; Section IV gives the overview of Argus. Sections V and VI elaborate on the design of EEPS and Argus respectively. Section VII describes the implementation of Argus in a mica2dot mote. Section VIII shows how the design of EEPS and Argus, and the implementation of the latter, achieve our goals of bandwidth management. Section IX briefly describes details of the testbed and other related components in the network stack. Results of experiments run on the testbed are presented in Section X. Next, we discuss some related issues encountered during the course of the experiments in Section XI, and finally we conclude in Section XII.

II. Related Work

The control of route-through traffic has been studied extensively in wired networks. For instance, Weighted Fair Queuing (WFQ) [2] maintains a separate queue for each flow passing through an Internet router, determines when a packet will finish being serviced bit-by-bit, and schedules them for transmission based on their virtual finishing times. Although such a scheme provides max-min fairness for each flow, it is impractical for sensor networks for three reasons:

1. Sensor motes have limited memory. Maintenance of a queue for each downstream mote is impossible for networks larger than a few motes since each mote only has about 4Kilobytes of RAM.
2. Simulation of bit-by-bit packet servicing consumes CPU cycles, which is limited and should be reserved for applications as far as possible.
3. Knowledge of each flow's weight requires communication overhead that consumes scarce energy resources.

A more scalable technique to approximate FQ within a network is proposed in Core-Stateless Fair Queuing [3]. Here, per-flow management is handled only at the edge routers, whereas the core routers compute the forwarding probability based on the current estimated rate inserted in the packet header. Thus core routers need not maintain per flow state, allowing the network core to approximate FQ at a much lower cost.

WFQ in wired networks has been extended to the wireless domain in [7], where the authors proposed a global and a distributed local fairness model. However, maintenance of each flow's state is still required, which may still be intractable for large sensor networks with multiple flows.

The Adaptive Rate Control (ARC) scheme [5] developed for sensor networks that gives preference to route-through traffic. By estimating the number of downstream motes, local rate metering can be performed to achieve a certain level of fairness. In general, usage of ARC causes more packets to be received from downstream motes. Also, since ARC does not maintain state pertaining to the flows, it cannot allocate more bandwidth to a particular flow in general.

III. Bandwidth Management Goals

In this section we describe precisely what the bandwidth management component aims to achieve.

1) Argus should handle various types of traffic, including many-to-one and point-to-point, in-network processed and aggregated traffic. It should also be able to take into account traffic that is periodic, that is triggered on some event, and that transfers data as fast as possible through the network. Since we focus on data traffic in this paper, we do not consider control traffic that are broadcast throughout the network.
2) It provides a common standard with which a traffic flow can use to determine the amount of bandwidth it obtains relative to other flows *when the network is congested*. For instance, a flow requesting for 4 units of bandwidth will obtain twice as much as another requesting for 2 units. We do not provide *absolute* bandwidth guarantees, that is, even though applications can attempt to send periodically at 10kbps, the actual bandwidth obtained will be subject to the amount available. Also, in general, it is difficult to guarantee a fixed amount of bandwidth with the wireless link quality fluctuating. Thus, we

upper-bound the bandwidth obtained, but not lower-bound it.

3) When the network is not congested, motes are allowed to send packets at their current rate. Transition of the network to and from a congested state and the corresponding handling of traffic should be *transparent* to the application. Thus the application need only be concerned with whether or not the network can accept more packets, and can otherwise send packets at any interval.
4) It performs some amount of *congestion control*, by regulating the rate at which packets are injected into the network, i.e. via *admission control*.
5) Finally, our mechanism allows the application to determine the division of bandwidth between downstream motes. This enables the application programmer to adjust the distribution according to his needs. Thus, with information from the MAC layers, differences between the maximum capacities of incoming links can be factored in. Using the earlier example on micaz and mica2dot motes, we give approximately 90% of the outgoing bandwidth to the incoming micaz packets, and the remaining 10% to the mica2dot packets. We focus only on the mechanism, the policies involved are beyond the scope of this paper.

IV. Overview of Argus

We begin by providing a high-level view of Argus, then elaborate on its design and different components in the subsequent sections.

Argus primarily manages the bandwidth obtained by a flow of data through the network. We assume that the bandwidth consumed by control traffic is small and therefore do not consider it further in this paper. Basically, each mote in the network monitors its current effective transmission rate and the total size of the flows routed through it. Dividing that rate by the total flow size gives the rate at which each unit of flow downstream can be injected into the network. The inverse of this rate gives the epoch length which is then propagated downstream, where each mote then regulates the number of packets allowed into the network, i.e. admission control is enforced locally.

When congestion is not experienced, Argus does not constrain any data flow, thus the applications can send at their desired rates. When congestion occurs, Argus can, by scheduling per-child queues and using the number of flows routed through each child, divide the available bandwidth between the flows as specified by their weights. This is done without explicit signaling between downstream and upstream motes, and is achieved with minimal overhead, in the form of piggybacked epoch length and total flow sizes, in the headers of data packets.

Argus handles two primary types of traffic rate patterns. The first type is *application determined* (AD). Examples of this kind of traffic include the periodic generation of data packets, and data that is produced upon the occurrence of some event, i.e. the application determines when it will produce packets. It is possible that part(s) of the network transit between congested and uncongested states, in which Argus manages the flows in a manner transparent to the applications, which will only know whether a packet can be accepted by the network, or not.

The second type of traffic is *network determined* (ND). In this case the rate at which packets can be pushed into the network is determined by the network itself. Typically, an application that generates ND traffic has a backlog of packets which it has to send as fast as possible. For instance, a mobile mote may gather readings over a long period of time and store them in non-volatile memory, and download them only when it is within the range of some infrastructure, or fixed, motes. In such instances the download of data should take place as quickly as the network allows. Unlike AD traffic which may or may not congest the network, it is clear that ND traffic will definitely do so.

We next describe the design of the algorithm that allows us to achieve the goals mentioned in Section III.

V. Algorithm Design

We extend the basic algorithm, Epoch-based Proportional Selection (EPS) proposed in [8]. We first define the common terms used throughout the rest of the paper, then briefly describe the steps to be taken prior to EPS, and finally the algorithm itself.

A. Terminology

We define the *parent* of a mote to be the mote to which a packet is next forwarded, and a *subtree* to consist of a mote and all its downstream motes. Thus, in Figure 2, mote A's *subtree size* is 5, and D's is 1. Assume that mote A requests for 2 units of flow, B requests for 5, C requests for 1, D requests for 3, and E requests for 1, then the *subtree flow sizes* for mote A, B and E are 12, 9 and 1 respectively.

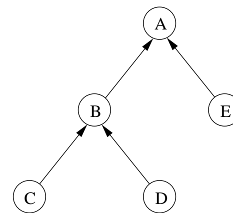

Fig. 2. Motes A, B and C have subtree sizes 5, 3 and 1 respectively.

B. Basic Steps

An outline of the steps to take is given below. We elaborate on each step in the following subsections.

1) Obtain the total units of flows n from downstream motes.
2) Measure the average rate r at which packets can be transmitted from this mote.
3) Divide the average rate r by total units of flow n to give the per-unit-flow generation rate $r_{flow} = \frac{r}{n}$.
4) Compare the rate r_{flow} with the rate $r_{flow,parent}$ sent from the parent mote, as well as with the rate $r_{flow,qfull}$

used when the local buffers exceed a predetermined threshold, use and propagate the smallest rate to the children motes.

5) Within an epoch $t_{epoch} = \frac{1}{r_{flow}}$, for a mote with f units of flows, inject no more than f packets into the network within the time interval t_{epoch}.

C. Obtaining Subtree's Total Flow

The subtree flow size of a mote A is stored in the header of the data packet sent to its parent mote B. At B, it retrieves the flow sizes from data packets sent by its children, sums them, and adds its own flow size before dispatching the total size to its parent in the next data packet. With reference to Figure 2, mote C sends a flow size of 1, B sends `SubTreeSize(C) + SubTreeSize(D) + 5` = 1+3+5 = 9, and A sends `SubTreeSize(B) + SubTreeSize(E) + 2` = 9+1+2 = 12.

Since the updated flow sizes are stored in all data packet headers, changes in these sizes can be updated and handled dynamically. This allows the network to adapt to changes in topology and in traffic flows. A new flow of size x increases the total flow size by x, and reinstates the previous value when it ends.

D. Rate Measurement

We now describe one way of measuring mean effective transmission rate at a mote. The effective time t taken to transmit a packet begins when the network layer sends a packet to the data-link layer, until the time notification of successful transmission is received from the data-link layer.

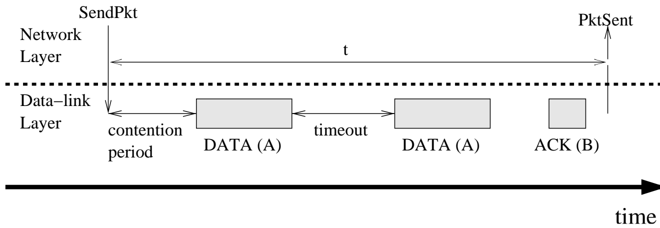

Fig. 3. SimpleMAC in operation

Using a simple MAC protocol, the operation of which is illustrated in Figure 3, t is then defined as shown in the figure. This value of t is a sample of the average time needed to transmit a packet, and this average time is obtained using the exponential moving average formula below:

$$av_i(t) \leftarrow \Big(\frac{\alpha}{n}\Big)T + \Big(1 - \frac{\alpha}{n}\Big)av_{i-1}(t)$$

where $av_i(t)$ is the moving average in the ith iteration, α is the weight, T is the latest sample of t, and n is the total number of subtree flows. The inverse of the average transmission rate gives the per-flow rate r_{flow}.

We note that this method of rate measurement takes into account loss due to corruption, interference and scheduling constraints. For instance, if S-MAC [6] puts the radio to sleep before sending the packet, the sleep time will be included in t, and thus correctly reflect the reduction in effective bandwidth.

E. Reaction to Overflowing Queues

When a mote detects the imminence of queues overflowing, we use the per-flow rate $r_{flow,qfull}$ which is adjusted according to Table I, where $r_{flow,parent}$ is the flow rate disseminated by the parent, motes begin reducing flow rates once the queue size increases beyond the `THRESHOLD`, and max_queue_size is the maximum occupancy amongst all queues in the mote.

TABLE I
FLOW RATE ADJUSTMENT: QUEUE OVERFLOW

```
if max_queue_size > THRESHOLD
   if queueFull == FALSE
      queueFull ← TRUE
      r_flow,qfull ← min(r_flow, r_flow,parent)
   else
      r_flow,qfull ← min(r_flow, r_flow,qfull, r_flow,parent)
   r_flow,qfull ← (n/(n+α)) r_flow,qfull
else queueFull ← FALSE
```

F. Epoch Length Definition

The length of an epoch, t_{epoch}, is defined to be the inverse of $\min(r_{flow}, r_{flow,parent}, r_{flow,qfull})$. This is the value that will be used locally. For traffic that is forwarded unaltered, we disseminate the same t_{epoch} downstream. However, for flows that undergo network processing, the outgoing flow size may not be the same as the incoming one, thus a simple adjustment has to be made to t_{epoch} before dissemination: Let the outgoing flow size per epoch be $f_{outgoing}$, and that of incoming ones be $f_{incoming}$. Then, the effective epoch length is simply

$$t_{epoch,updated} = t_{epoch} \times \frac{f_{incoming}}{f_{outgoing}}$$

By controlling the epoch length, we manage the rate at which packets are pushed into the network, thereby controlling congestion. In general, a mote with a total flow of x units is allowed to push a maximum of x packets into the network within an epoch.

G. Dissemination of Epoch Length

Like the flow size of the subtree, the epoch length is piggybacked on the data packets transmitted. The children motes eavesdrop on the data packets, and update their parents' epoch lengths correspondingly.

H. Extended Epoch-based Proportional Selection (EEPS)

The basic Epoch-based Proportional Selection (EPS) algorithm transmits, within each epoch, exactly from each child's queue n times the number of downstream motes serviced by that queue, where n is some positive integer. Work conservation is not implemented: if a packet to be sent has not yet arrived, the mote waits for it. Also, since a mote does not keep track of packets from each downstream mote but the number of packets transmitted within an epoch, the queues must be FIFO. Once the two conditions above are met, EPS is correct based on the following inductive proof:

Base case: Consider a leaf mote L. Within an epoch, L will transmit exactly one packet from itself, equal to the number of motes in its subtree.

Inductive step: Assuming that, exactly one packet from each downstream mote arrives in the corresponding FIFO queue within an epoch, then, by transmitting the same number of packets as the number of downstream motes, EPS ensures that exactly one packet in its subtree will be transmitted.

The extension of basic EPS is straightforward: for a mote generating x packets per epoch, we treat that mote as a logical mote consisting of x physical motes, and the correctness proof can be applied as before.

VI. ARGUS DESIGN

We describe how EEPS can be applied to a network with varying traffic flows and patterns, as well as the components that its implementation, Argus, is composed of.

A. Per-Epoch Management

The time granularity used in Argus is the epoch length as defined in Section V-F. When the system is relatively stable, packets at depth d in the nth epoch will be sent to motes at depth $(d-1)$ in the $(n+1)$th epoch. Since traffic flows may vary over time, we handle them on an epoch-by-epoch basis. The primary issue becomes the determination of flow sizes within an epoch, which is easily solved with the observation that the flow size is a combination of that reported by packets from children motes and the number of packets generated locally in the previous epoch.

B. Per-Destination Queue Management

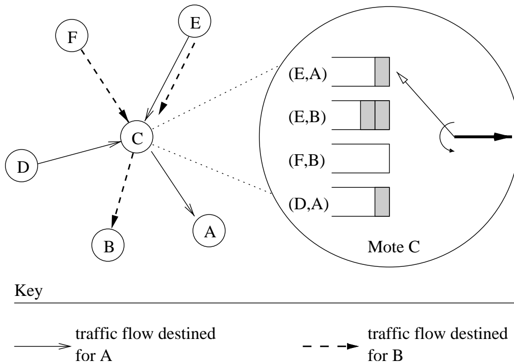

Fig. 4. Cross traffic: mote C routes two different traffic streams: one from D and E to A, and another from F to B. In C, (E,A) refers to the queue associated with child E and destination A. Scheduling is performed amongst all queues.

Since a mote may be routing packets to different destination motes, queues have to be allocated per destination as well. In Figure 4, mote C routes traffic from D and E to A, and from F to B. Mote C treats each child queue as distinct queues, and schedules them together rather than perform scheduling amongst queues grouped according to their destinations. The former is done because Argus aims to allocate bandwidth to *all flows* routed through a mote according to its weight relative to the other flows, regardless of their destinations. Although per-child, per-destination queues require more memory in resource constrained motes, we believe that this is acceptable for sensor networks as the number of sink motes in the network at any time is expected to be small. Also, queues corresponding to a particular sink can be reused if no more traffic is routed to it.

C. Weight Management

Flow sizes associated with each queue can be thought of as weights, which can then be manipulated to provide control over the bandwidth distributed amongst the queues. Referring again to Figure 4, mote C may allocate twice the bandwidth to packets from E by doubling the weights of E's queues. Alternatively, increasing the weights for queues associated with destination A will increase the rate at which packets will be sent to A. Thus, weights can also be altered to take into account the capacities of the data-link layers. It is to be noted that this adjustment of weights is performed locally, though one can possibly construct a control plane that globally manages flows. This, however, is beyond the scope of this paper.

D. Flow Management

A critical aspect of EPS is the *knowledge that a packet from a child is supposed to be sent next, and to wait for it before starting the next epoch*. In EPS, it is assumed that flows are continuous, and that each mote generates a data packet every epoch. Since the length of an epoch is determined by the current network state, EPS is suitable for motes generating *ND* traffic. To extend EPS to incorporate *AD* traffic, we observe that a mote (say A) needs only indicate whether its parent, as well as subsequent upstream motes, are required to *anticipate* more packets from A. The conditions under which the parent mote needs to do so are:

- A's queues become backlogged, in which case A will definitely have more packets to send, and
- A routes ND traffic, which is pushed into the network whenever the network is able to accept more.

We use one bit (the "more" bit), to indicate whether more packets are due to arrive, and set the bit according to the conditions stated above. Once the "more" bit is set, it will not be reset before reaching the destination.

With the knowledge that more packets will arrive, and because flow sizes may differ between consecutive epochs, Argus waits until all anticipated packets, carrying the sizes of flows for the next epoch, to arrive before continuing.

We use a simple example illustrated in Figure 5 to explain the use of the "more" bit. Assume that motes S1 and S2 initially send at 1 packet per minute. At such a low rate, the paths from S1 and S2 to R via X are uncongested: the queues never have occupancy greater than one. Together with the fact that the traffic are not of type ND, this means that the "more" bit never gets set, and thus, at X, each packet can be sent as soon as they arrive, without having to wait for the other.

Now assume that S1 and S2 increase their sending rates, perhaps due to the detection of an event, to the point where

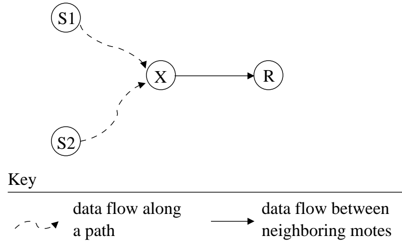

Fig. 5. Source motes S1 and S2 are sending packets to receiver R. Traffic from both sources are routed through X.

X can no longer forward packets as quickly as they arrive. At this time, X's queues, as well as those along the path S1-X, and S2-X, build up. The latter causes incoming packets at X to have their "more" bit set, thus X alternates between sending packets from S1 and S2, i.e. the system automatically reverts to EEPS.

Having described Argus' design, we next elaborate on its implementation in two separate components: Argus-T, and Argus-N.

VII. Argus Implementation

Argus' functionalities are implemented in the network (Argus-N) and transport (Argus-T) layers of the traditional network stack as shown in Figure 6(a). Whilst in reality it is possible to implement the functionalities in one component, we split them into two for clarity and ease of explanation.

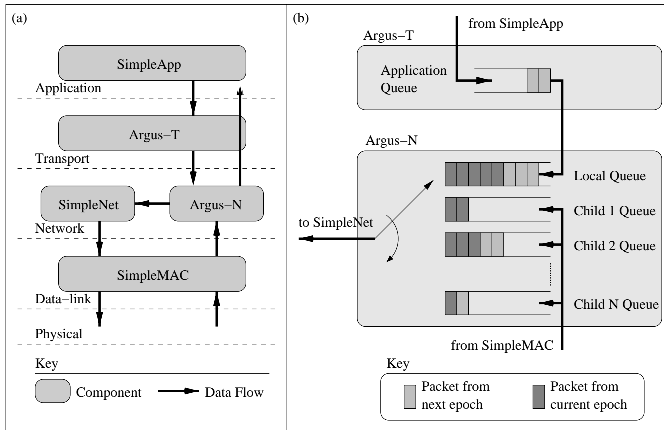

Fig. 6. (a) Location of Argus components (Argus-N and Argus-T) in the network stack. SimpleMAC, SimpleNet and SimpleApp are generic media-access control, forwarding and testing components respectively. (b) Per-child queues in Argus-N: a special local queue is reserved for packets generated locally, if any. These packets are transfered at the end of each epoch from Argus-T's application queue to the local queue in Argus-N.

A. Argus-N

Having the forwarding path traverse through Argus-N is necessary in order to control which packet to send next and when. Argus-N contains the queues used to buffer packets to be forwarded, including those generated locally. The variables `parentEpochs` hold the updated epoch lengths of each parent mote corresponding to different destinations, whilst three variables are associated with each queue:

1) `currentFlowSize`: gives the number of packets yet to be transmitted in the current epoch,
2) `maxFlowSize`: maintains the total number of packets to be transmitted in the current epoch, and
3) `hasMore`: a boolean variable indicating whether more packets are anticipated from the associated child mote,

With reference to Figure 6(a), six operations are performed, triggered by the following events:

Reception of data packet: An incoming packet is buffered in the corresponding queue based on the previous one hop mote from which it arrived as well as on the destination mote as specified in the packet header. The "more" bit in the packet header is also copied and updated in the corresponding `hasMore` variable.

Selection of next packet to send: Selection is performed in a round-robin manner amongst queues that have packets to send in the current epoch. If there are packets to send that has not yet arrived (i.e. we sent less than the flow size thus far for this epoch), and there are no other queues that can be serviced, then we wait until the packet arrives, *work conservation is not implemented*. If no more packets are to be sent in this epoch, then the next epoch begins, with `currentEpochSize` updated from the first packet in each queue. However, if `hasMore` indicates that packets are expected from a child and none has yet to arrive, then the next epoch does not begin until at least one packet from that child is received. All waiting times are included in the effective transmission time of a packet.

Successful transmission of packet: The transmission time begins when the next packet is sent or should have been sent. Computation of effective transmission time, as specified in Section V-D, is performed when the data-link layer signals the successful transmission of a packet.

Reception of eavesdropped packet from parent: The epoch length piggy-backed on the packet header is retrieved and stored in the corresponding `parentEpoch` variable.

Ending of an epoch: When an epoch ends, Argus-N computes the duration of the next epoch according to Section V-F. This duration is jittered to prevent synchronized injection of packets into the network by the application [5].

Sending of the next packet: Just before the next packet is sent, the packet header is updated with the current epoch length, and the sum of the flows routed through this mote to the particular destination.

B. Argus-T

Argus-T interfaces with the application, and provides admission control. The application first informs Argus-T of its intention to send and receive packets using the function call

```
opensocket(application_id,
           flow_size,
           destination_address,
           intercept,
           network_determined)
```

The parameters are explained as follows:

- *application_id*: is the identification number of the application, since in general we may have different applications running simultaneously in the network.
- *flow_size*: for AD traffic, `flow_size` indicates the maximum number of packets the application will send per epoch, and the number of packets sent from this flow if and when congestion occurs. If the traffic is of type ND instead, then `flow_size` refers to the number of packets the application is expected to send per epoch.
- *destination_address*: the identification number of the destination mote.
- *intercept*: a boolean variable. If true, packets received with the same `application_id` will be passed to the application for processing. This is useful for applications that perform in-network processing of data, for instance, data aggregation.
- *network_determined*: a boolean variable. If true, the rate at which the application sends packets is determined by the network. Also, the start of an epoch will be signaled to the application, which can then proceed to send *flow_size* number of packets for that epoch.

Argus-T controls admission of packets into the network by limiting the number that can be injected within an epoch, the length of which is determined by Argus-N in the previous section. Packets accepted are not immediately pushed onto the local queue in Argus-N, but buffered within Argus-T until the next epoch begins. This buffering is necessary in order to precisely determine local traffic's flow size for the next epoch. The number of packets accepted per epoch is upper bounded by the parameter `flow_size` set by the application when it calls `opensocket`.

Like Argus-N, operations are typically performed upon the triggering of specific events:

Reception of a packet from application: Argus-T first checks if the application has exceeded its quota for the current epoch, and that there is sufficient local buffer space. If so, the packet is accepted.

Ending of an epoch: Packets buffered in the current epoch is pushed into the local queue in Argus-N. As each packet is transferred, the number of packets remaining in the local buffer is copied into the flow size field of the packet header. Logically, Argus-T is similar to a child mote forwarding packets to its parent.

VIII. ACHIEVEMENT OF GOALS

With the implementation details given in Section VII, we now show how the goals set out in Section III are met.

Different traffic types: For unaggregated traffic, where each intermediate mote forwards incoming packets (not due to itself) without processing, `opensocket` is called with parameter `intercept` set to false. Also, for such traffic the total size of the flow as seen by the destination mote is simply the sum of flows of all motes in its subtree. Figure 7(a) shows the path of the data flow within the intermediate mote.

For packets that have in-network processing operations performed on them, `opensocket` would be called with

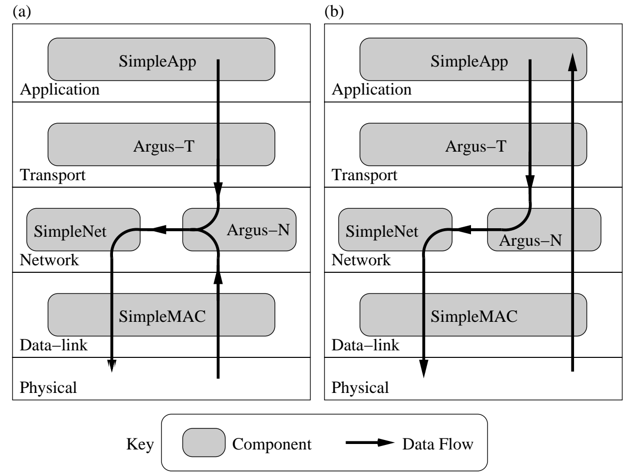

Fig. 7. (a) datapath within an intermediate mote that does not perform in-network processing of received data, and that generates data of its own. (b) datapath within an intermediate mote that performs in-network processing.

`intercept` set to true. The data path is given in Figure 7(b), and both incoming and outgoing per-epoch flow sizes have to be maintained to handle traffic that is processed in-network (Section V-F).

As mentioned in Section IV, Argus also handles AD and ND traffic. For AD traffic, `opensocket` is called with `network_determined` set to false. At low data rates when the queues are relatively empty, "more" bits in the data packet headers will not be set most of the time; furthermore, flow sizes at anytime are likely to be small and thus packets can be forwarded as soon as possible without having to wait for packets from other queues. On the other hand, when the network becomes congested, the system will revert to EEPS as specified in Section V-H.

As for ND traffic, the "more" bit in packets carrying this kind of traffic is always set. Unlike the case of AD traffic, control over sending of packets is passed to the network, which signals the application when the next epoch has began and packet(s) can be accepted.

Congestion control: Congestion control is implemented in various parts. Firstly, the epoch length disseminated downstream (by Argus-N) by a congested destination or intermediate mote is used when performing admission control (by Argus-T) of packets into the network. Secondly, as mentioned in [8], the continuous feedback and adaptation of the network to the current environment brings about oscillations in the epoch length, thereby causing phase shifting between motes of different depths with respect to the destination mote. Thirdly, jittering is introduced when computing the length of the next epoch, which determines the time locally generated packets are injected into the network. This provides further shifting of phase independent of the application's behavior.

Common standard: The `flow_size` parameter in `opensocket` can be used to determine the relative amount of bandwidth a flow obtains with respect to another

during congestion. The range and typical application-specific values for `flow_size` can be determined by an out-of-band mechanism, much like the assignment of TCP ports to well-known applications in the Internet.

Transparent state switching: This goal is relevant only for AD traffic, since ND traffic always attempt to congest the network. To the application, it does not need to know if the network is congested, or to be concerned with the fairness of the network and thus adjust its rate accordingly. The application needs only know whether a packet can be accepted or not. The setting of "more" bits, which signify an increase in occupancy and therefore possible congestion, is transparent to the application.

Dynamic altering of bandwidth allocation: The weights assigned to each queue is derived from the corresponding flow sizes piggy-backed on each data packet. By altering these weights, we can alter the distribution of outgoing bandwidth amongst the incoming queues. This can take into account differences in MAC layer technologies, and also allow applications to meet certain requirements. For instance, we can increase the weights of a subtree of motes in a certain region of the rainforest, to obtain more readings from them over a period of time. An alternative would be to contact each of the motes and get each of them to increase their flow sizes. The former is a more attractive option as it reduces the communication overhead that is required in the latter.

Altering of bandwidth management is also possible at the application end to try and obtain an absolute, rather than relative, amount of bandwidth in the network. The application can occasionally monitor the rate at which it has been sending packets, and adjust the `flow_size` parameter accordingly. Since it is possible that the bandwidth requested is not available, `flow_size` must be upper bounded to prevent the application from increasing that to a large value.

IX. TESTBED DETAILS

In this section we briefly describe the implementation of the other components in the network stack: SimpleMAC and SimpleNet, the application SimpleApp, and details of the testbed used.

A. Testbed Description

We tested our implementation on a mica2dot testbed [9] of 26 motes. Each mote is connected via its serial interface to a 100Mbit ethernetwork, which is in turn connected to a server from which we can upload binaries via the ethernetwork and serial interface. The ethernetwork / serial interfaces also allow us to log data packets in the server just before they are sent from the source motes, and when they are received by the destination mote; it is not used for actual data communication between motes. The ethernetwork is completely isolated from other networks, thus traffic consists only of packets sent to and from the motes. The latency in the ethernetwork is low, on the order of a few milliseconds, an assumption that is verified in the next section.

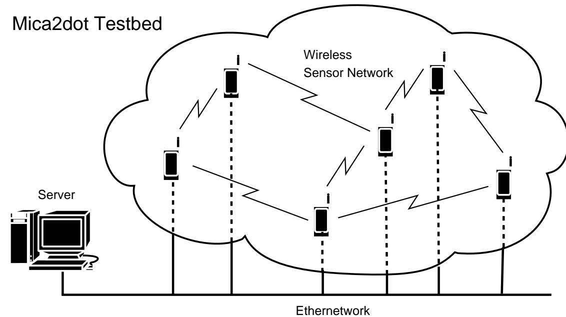

Fig. 8. The mica2dot testbed on which our implementation ran. Each mote is connected to an ethernetwork via the serial interface, and can communicate with a server. The ethernetwork is used to log packets sent and received from each mote.

B. SimpleMAC

We include a description of SimpleMAC, SimpleNet and SimpleApp for the sake of completeness. When a packet is passed to SimpleMAC, it waits for a random period of time (contention period). If nothing is heard from neighboring motes, it transmits the data packet and waits for an acknowledgment (ACK). If none is heard, timeout occurs, and it backs off before entering the contention stage again. Each retransmission exponentially increases the backoff period, and the relevant constants are given in Table II. Since Argus is supposed to be independent of the data-link layer, we kept SimpleMAC simple, and do not attempt to improve its throughput by optimizing the various parameters in the table. Also, SimpleMAC does not buffer packets: once a packet has been received, it is passed to Argus-N immediately. Similarly, a packet from SimpleNet is sent as soon as possible.

TABLE II

TABLE OF CONSTANTS: SIMPLEMAC

ACK timeout	25 ms
Contention period (max)	100 ms
Initial backoff period (max)	100 ms
Maximum backoff period (max)	400 ms

C. SimpleNet

In general, the topology of the multi-hoping network is a factor in the performance of our algorithm. To provide a fair comparison with round-robin queue scheduling, and because we are not concerned with how routing is done, we fix the topology by defining the next hop at every mote for a particular destination. The links in the resulting topology have relatively low loss, and are selected via an out-of-band mechanism. The topology is given in Figure 9. In general, the larger the mote id, the more hops it is from the destination.

D. SimpleApp

SimpleApp is an application written to generate and receive three different types of traffic: periodic, in-network processed, and ND. It also performs the additional task of notifying packet generation and packet reception at the destination mote via the serial interface.

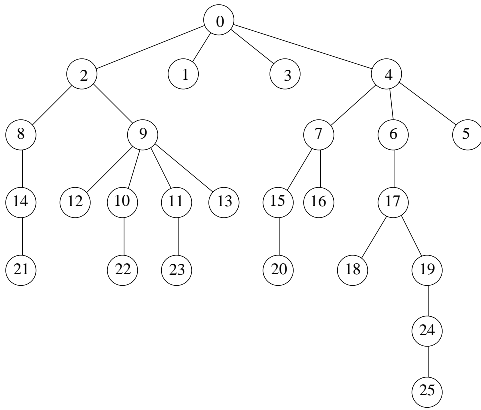

Fig. 9. Fixed routing topology: mote 0 is the destination and the root of the routing tree.

X. Empirical Results

In this section we present the empirical results obtained from our implementation of Argus on the testbed described in the previous section. The network topology as mentioned in Section IX-C is fixed (Figure 9) to factor out effects that may otherwise result. Throughout this section we compare Argus with the round-robin servicing of queues, and we begin by providing motivation for Argus.

A. Motivation

We first vary the rate at which packets are generated by SimpleApp. Figure 10 shows the fraction of packets received at the destination that originate from the corresponding mote. At a sending interval of 5 seconds for each mote, the network is still uncongested and is able to deliver the packets at the same rate as they are generated. However, when the application attempts to send at intervals of 3.75 and 2.5 seconds, the network is unable to handle the load and in general becomes biased towards motes closer to the destination. Thus, each mote has no control over the amount of bandwidth it effectively receives when the network becomes congested.

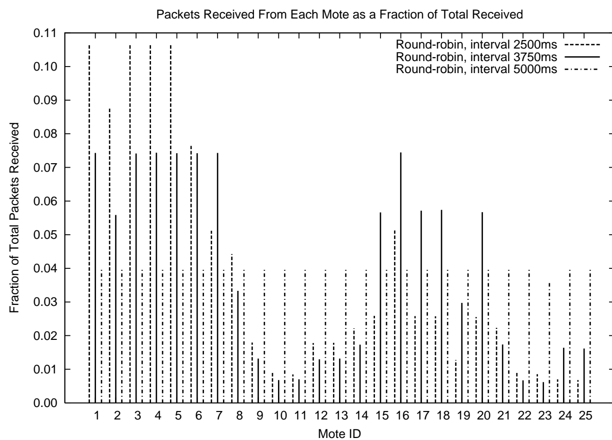

Fig. 10. Motes with smaller IDs are in general closer to the destination mote. More packets are typically received from motes closer to the destination as the network gets congested.

B. Periodic Data Generation

We first compare the performance of Argus and round-robin with SimpleApp generating packets at 3.75 second intervals. Periodic generation of data is a form of AD traffic. In the case of Argus, each flow in the network is given a size of two that remains unchanged for the duration of the experiment, which lasted for an hour. The results in Figure 11 clearly shows that for periodic traffic, Argus is able to evenly distribute available bandwidth amongst all downstream motes.

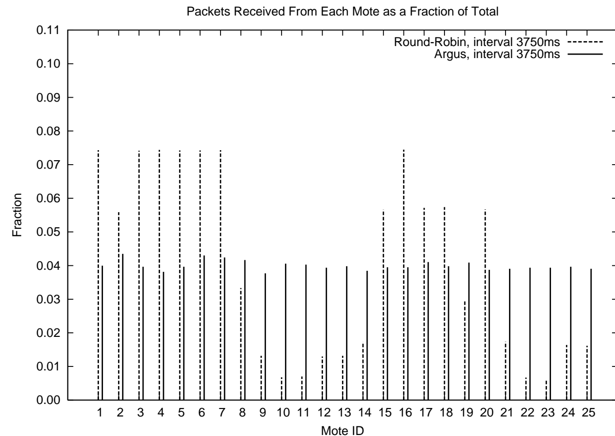

Fig. 11. With motes attempting to send at intervals of 3.75 seconds, Argus evenly distributes bandwidth amongst all motes regardless of their hop distance from destination.

C. Latency

We next investigate the delay experienced from the time a packet is first injected into the network until the time it reaches the destination. This delay includes the time the packet takes to traverse the ethernetwork before being logged in the server. We believe that the processing time within the server is negligible, and thus only investigate the delay within the ethernetwork by pinging the motes for the duration of an experiment. Round-robin pinging of the motes is performed, the interval between consecutive motes being exponentially distributed [1] to ensure that the correct average delay values are obtained. Also, the intervals have a mean of two seconds so that the ping packets themselves do not congest the ethernetwork. The results are given in Figure 12, which shows that the delay is small, averaging about 2 milliseconds, and does not vary significantly even when the wireless sensor network is congested.

Figures 13 and 14 show the latency distribution for round-robin and Argus respectively. We observe that Argus' delay distribution has less variance compared to round-robin's. This is despite the fact that SimpleMAC is a random access protocol and is therefore difficult to control the latency across multiple hops. Another observation is that the latency becomes independent of the number of hops the mote is from the destination. In Figure 13 we find that motes that are closer to the destination have their packets delivered faster than the others, whereas the network is more fair in the case of Argus: packets experience roughly the same delay.

An intuitive explanation behind why Argus causes a reduction in mean and variance in latency is that the upstream

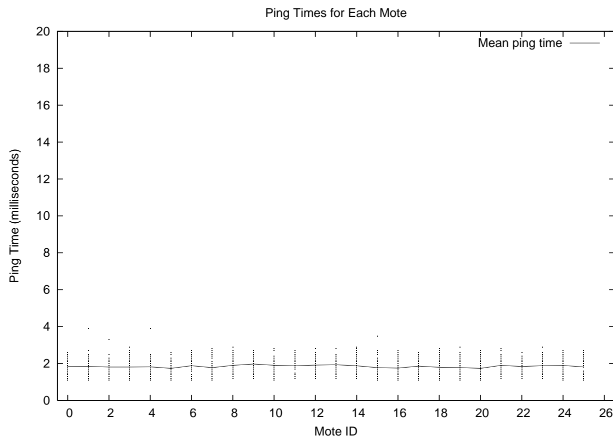

Fig. 12. Distribution of time to ping each mote on the ethernetwork, with the mean times.

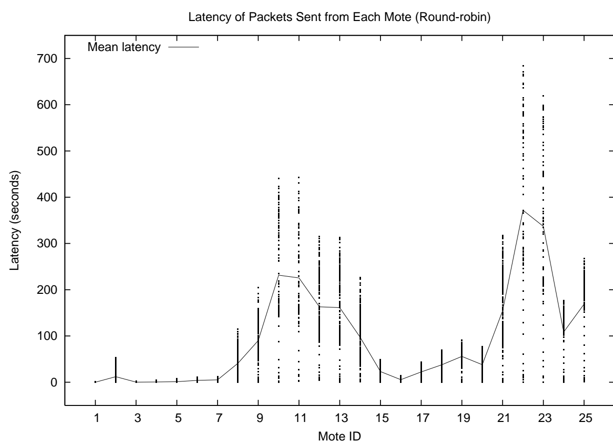

Fig. 13. Latency Distribution and Mean for Packets Generated by Corresponding Motes - Round-robin

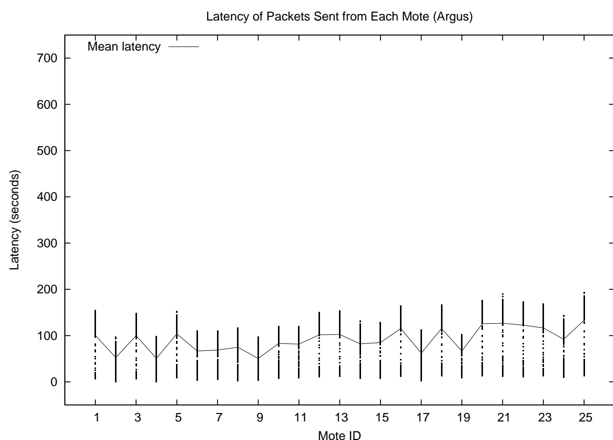

Fig. 14. Latency Distribution and Mean for Packets Generated by Corresponding Motes - Argus

motes know and expect packets from downstream motes. If these expected packets do not arrive, then upstream motes keep silent, which increases the chances of successful reception and consequently packets originating downstream are more likely to be forwarded.

D. Transition from Uncongested to Congested State

We next demonstrate that Argus is able to transparently handle the distribution of bandwidth as the network transits from an uncongested to a congested state. Initially each mote in the network transmits periodically at an intervals of 60 seconds. After 30 minutes, the motes increase their generation rates to send at intervals of 3.75 seconds. Throughout the entire process, SimpleApp is not notified of any changes in the network.

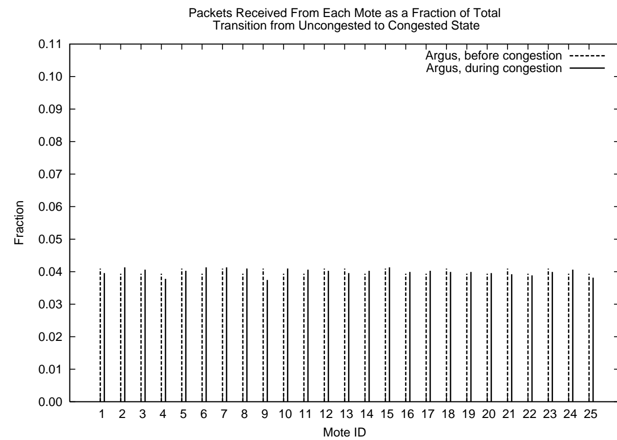

Fig. 15. Fraction of total packets received from each mote is approximately equal before and after the network becomes congested.

We gathered the fraction of packets received from each mote before and after the increase in generation rates. The results are given in Figure 15, which shows that Argus achieves its goal of transparently handling network state transitions.

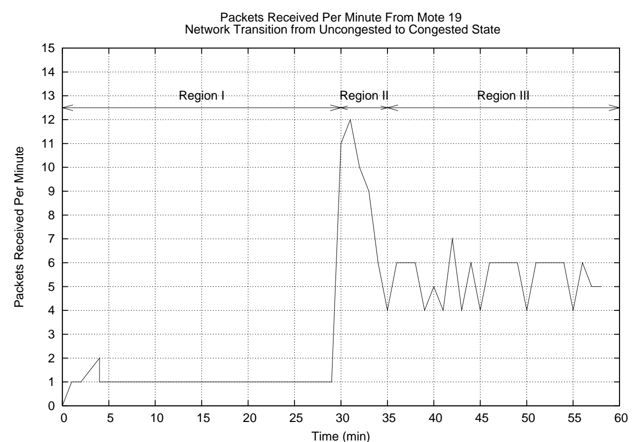

Fig. 16. Rate at which packets are received by the destination mote before and after the network becomes congested.

We also monitored the per-minute rate at which the destination mote received packets from mote 19 (Figure 16), as well as the fluctuation in its epoch length at the same time (Figure 17). We split the graphs into 3 different regions, I, II and III, and elaborate on each as follows:

Region I: Epoch lengths are initialized to a large value, 20 seconds, to prevent immediate congestion of the network

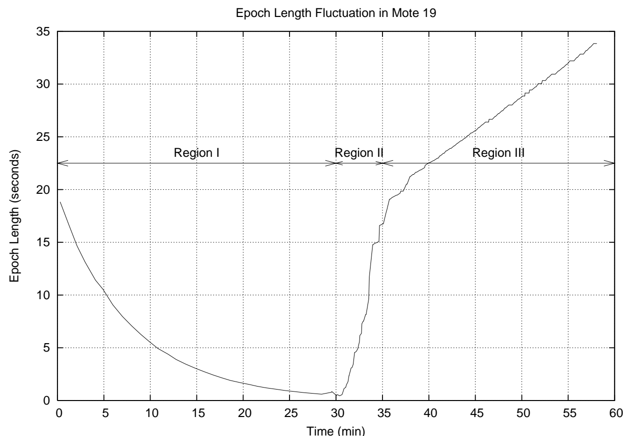

Fig. 17. Epoch length fluctuation: the network transits from an uncongested to a congested state.

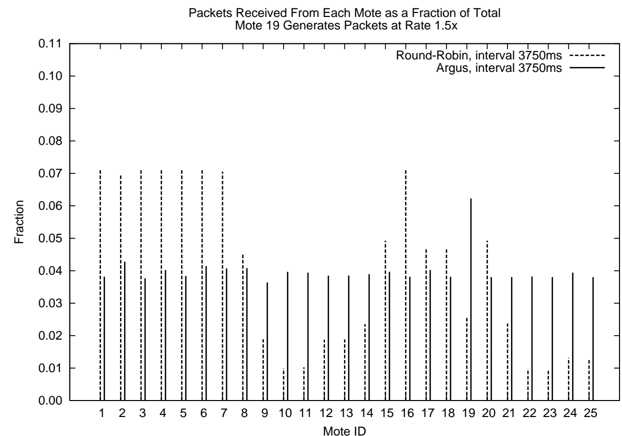

Fig. 18. Mote 19 requests for 1.5 times the bandwidth of other motes. Argus realizes this allocation, but not round-robin.

if generation rates are high. This is similar to TCP's slow start [4]. The network is relatively unloaded and each mote generates packets at the fixed rate of one packet per minute. Since the network is uncongested packet collisions are uncommon and the effective transmission time of a packet is close to the actual time to transmit one (in the tens of milliseconds). This causes a drop in the epoch length, and since an exponential weighted moving average is used in computing the length this results in the concave graph in Figure 17.

Region II: About 30 minutes into the experiment all motes begin generating packets at intervals of 3.75 seconds. Since the epoch length has become short by this time, packets are rapidly pushed into the network, causing the spike around the 30 minute mark in Figure 16. With the increase in rate, collisions occur, reducing the effective time to transmit a packet, and consequently the epoch length increases as well. Since the rate at which packets are transmitted is higher than in Region I, epoch length updates occur more frequently, and thus the increase in epoch length is more rapid than the decrease in Region I. This behavior is ideal since this is precisely the time when the epoch length should rapidly increase to reduce congestion.

Region III: During this time the number of collisions decrease, while the queue sizes increase. This causes the epoch length to keep increasing (computed according to Table I) in order to reduce the size of the queues. Although not shown in the figure, subsequently the epoch length decreases as queues drain, thereafter it oscillates around the 30 second mark, in response to the queue occupancies.

E. Non-uniform Bandwidth Distribution

Finally, we increase mote 19's generation rate to 1.5 times that of the other motes. For Argus, we increase mote 19's flow size accordingly, from 2 to 3. The result shown in Figure 18 shows that Argus is able to provide that extra amount of bandwidth to mote 19, whilst in the case of round-robin the increase in rate at the mote has no effect on the effective bandwidth it receives as seen at the destination mote.

XI. Discussion

In this section we discuss various related issues that were encountered during the design and implementation of Argus.

A. State

Due to the severe resource constraints in a sensor mote, the amount of state required by the algorithm has to be minimized as much as possible. An implementation of the network stack components should always maximize the amount of remaining resources for the application that it provides services for.

With this in mind, we note that the primary sources of memory consumption in Argus are the per-destination, per-child queues in Argus-N, and the application queue in Argus-T. Whilst it is possible to reduce each of the queue sizes in Argus-N to one, we cannot do so for the case of Argus-T, since, to determine the number of packets injected into the network (and thus the flow size in this epoch), we need to buffer them before transferring them to Argus-N's local queue in the next epoch. Thus, the application queue must be able to hold as many packets as the parameter `flow_size` specified by the application when it calls `opensocket`.

Having per-destination queues also imply that Argus will probably not be feasible for networks that have many sinks and long flows. For short flows we can implement timeouts: if a mote has not forwarded any packet to a particular destination and is not expecting to (via the `hasMore` variable), then the set of queues corresponding to that destination can be reused.

B. Timeouts and Packet Losses

As mentioned in Section IX-C, the network routing topology is fixed for all experiments to factor out effects that may otherwise arise. In reality, link qualities fluctuate causing relatively frequent changes in topology. Since state is stored with respect to a downstream mote in the form of the `hasMore` and `currentFlowSize` variables, there is a need

to anticipate and handle situations where the next expected packet doesn't arrive. This is possible since although link-level retransmissions can reduce loss probability due to interference and corruption, it cannot eliminate it completely. When the data-link layer fails to successfully transmit the packet within a number of attempts, the network layer may decide to (1) drop the packet and send the next, or (2) pick a different neighbor and attempt to forward the same packet again. In (1), dropped packets in the middle of a flow can be handled at the next hop mote by updating `currentFlowSize` if it is smaller, otherwise the packet is assumed to be from the next epoch. The problem with this approach is that the network then again becomes biased towards motes closer to the destination, albeit not as much as when Argus is not used. If however, the dropped packet is at the end of a flow, the next hop mote will then wait for its arrival, an event that will never occur. In this case, there is no alternative but to implement timeouts. In (2), the situation in the previous next hop mote is the same as the dropping of the last packet in the flow for (1): it waits for a packet that will never arrive. In this case timeouts will probably also have to be implemented.

Thus, we see that for optimal performance the routing component has to satisfy the following properties:
(1) it has to minimize losses by selecting next hop links that have low loss probabilities, and
(2) it has to minimize the frequency of changes in the routing topology.

C. On Topology Changes and Feedback

Consider a typical cycle in a stable network: the congested mote (say A) measures the effective transmission rate and disseminates the resulting epoch length. The epoch length is used in Argus-T when enforcing admission control, causing a reduction in packets pushed into the network. Subsequently, congestion at A reduces, and effective transmission time as well as the epoch length decrease. This forms a feedback loop that causes the epoch length to approach then oscillate around some optimal value. It can be observed that this convergence time increases with the path length of this feedback loop, i.e. the more hops the loop covers, the longer it will take to converge. Bearing this in mind, it is possible that, to take advantage of higher quality paths, the network topology changes at intervals that are significantly shorter than the time feedback takes to traverse the network. The moving average weight, α in Section V-D, also affects the rate at which congestion control converges. Exactly how to determine the optimal rate at which topology can or should change, and the optimal value for α, is an open question and will be included in future work.

D. MAC Optimization

We implemented a SimpleMAC to demonstrate that Argus is independent of the data-link layer. However, one can suggest optimizations in the MAC protocol to improve performance. For instance, we expect that one of the major causes of packet drops is the overflowing of queues caused by the wait for another child mote's (say X) packets. An optimization in this case, shown in Figure 19, would be to explicitly notify the children that the queues buffering their packets are overflowing, and to backoff for a longer period of time $t_{backoff}$ that is reasonably lower bounded to allow X to transmit its packet. $t_{backoff}$ can be a function of the number of one-hop children motes, so that the probability of interference does not increase with an increase in children motes but remain relatively constant. Such explicit notification has the advantage of distinguishing between loss due to corruption / interference and overflowing queues.

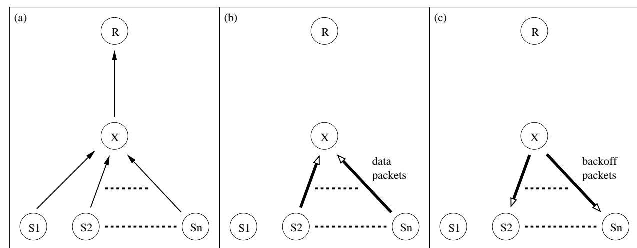

Fig. 19. (a) Mote X forwards packets from S1, S2, ..., Sn to R (b), (c) To increase the chances of successfully receiving the next packet from S1, X tells S2, ..., Sn to backoff for longer periods of time.

E. Interference Between Neighboring Flows

Whilst Argus explicitly manages cross traffic at a mote through the use of per-destination queues, it is less clear how traffic that do not cross but still interfere with each other should be handled. For instance, in Figure 20, where A and B route traffic to different destinations and are able to receive each other's packets, one may let the epoch length be the mean of that in A and B, or pick the larger of the two. On the other hand, if A and B are close enough to interfere with each other's transmissions, but are largely unable to receive the other's packets, then a solution will possibly require global coordination. These are areas that will be considered in future work.

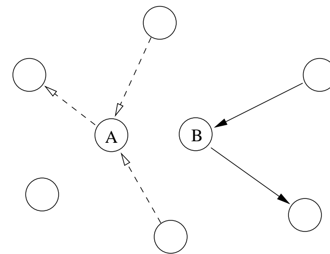

Fig. 20. Two different traffic streams exist in the network: one passes through A, another through B. A and B's transmissions interfere with each other.

XII. Conclusion

In this paper we presented a distributed, WFQ-like scheduling algorithm, Extended Epoch-based Proportional Selection (EEPS). EEPS requires per-child, per-destination queues and not per-flow queues, and is thus more scalable than WFQ. Weights associated with each queue is first derived from the sum of flows of downstream child motes, and may be altered to

take into account application requirements or MAC protocol differences. EEPS also measures and uses the length of an epoch to upper bound the maximum number of packets a mote can inject into the network, thus providing admission control that aids in reducing congestion. The implementation of EEPS, called Argus, is tested on a 26-mote testbed. Empirical observations show that Argus is able to reduce the mean and variance of the latency experienced by packets from motes several hops away from the base station. It is also shown that Argus is able to distinguish between flows of different sizes, and divide the bandwidth accordingly. Lastly, Argus transparently handles the transition of the network to and from a congested state, and can accommodate different types of traffic including Application-Determined (AD), Network-Determined (ND), many-to-one, point-to-point, and in-network processed.

REFERENCES

[1] R.L. Wolff, *Poisson Arrivals See Time Averages*, Operations Research 30, 223-231

[2] A. Demers, S. Keshav, S. Shenker, *Analysis and Simulation of a Fair Queuing Algorithm*, Proceedings of ACM SIGCOMM '89, pp3-12.

[3] Ion Stoica, Scott Shenker, Hui Zhang, *Core-Stateless Fair Queuing: A Scalable Architecture to Approximate Fair Bandwidth Allocations in High Speed Networks*, SIGCOMM 1998.

[4] M. Allman, V. Paxson, W. Stevens, *TCP Congestion Control*, RFC2581, April 1999.

[5] Alec Woo, David E. Culler, *A Transmission Control Scheme for Media Access in Sensor Networks*, Seventh Annual International Conference on Mobile Computing and Networking, pp 221-235, July 2001.

[6] Wei Ye, John Heidemann, Deborah Estrin, *An Energy Efficient MAC Protocol for Wireless Sensor Networks*, In Proceedings of the 21st International Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOMM 2002), pp 1567-1576, June 2002.

[7] H. Luo, S. Lu, V. Bharghavan, J. Cheng, G. Zhong, *A Packet Scheduling Approach to QoS Support in Multihop Wireless Networks*, ACM Journal of Mobile Networks and Applications (MONET), Vol. 9, No. 3, June 2004

[8] Cheng Tien Ee, Ruzena Bajcsy, *Congestion Control and Fairness for Many-to-One Routing in Sensor Networks*, Second International Conference on Embedded Network Sensor Systems (SenSys), Nov 2004

[9] *<blanked>*, http://<blanked>