PIP: A Connection-Oriented, Multi-Hop, Multi-Channel TDMA-based MAC for High Throughput Bulk Transfer

Bhaskaran Raman
Dept. of CSE, IIT Bombay
Powai, Mumbai, India 400076.
br@cse.iitb.ac.in

Kameswari Chebrolu
Dept. of CSE, IIT Bombay
Powai, Mumbai, India 400076.
chebrolu@cse.iitb.ac.in

Sagar Bijwe
Dept. of CSE, IIT Bombay
Powai, Mumbai, India 400076.
sag.bijwe@gmail.com

Vijay Gabale
Dept. of CSE, IIT Bombay
Powai, Mumbai, India 400076.
vijaygabale@cse.iitb.ac.in

Abstract

In this paper, we consider the goal of achieving high throughput in a wireless sensor network. Our work is set in the context of those wireless sensor network applications which collect and transfer bulk data. We present PIP (Packets in Pipe), a MAC primitive for use by the transport module to achieve high throughput. PIP has a unique set of features: (a) it is a multi-hop connection oriented primitive, (b) it is TDMA-based, (c) it uses multiple radio channels, and (d) it is centrally controlled. This represents a significant shift from prior MAC protocols for bulk data transfer.

PIP has several desirable properties: (a) its throughput degrades only slightly with increasing number of hops, (b) it is robust to variable wireless error rates, (c) it performs well even without any flow control, and (d) requires only small queue sizes to operate well. We substantiate these properties with a prototype implementation of PIP on the Tmote-Sky CC2420-based platform. PIP achieves about twelve times better throughput than the state-of-the-art prior work, over a network depth of ten nodes.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Wireless communication

General Terms

Design, Experimentation, Performance

Keywords

Bulk data transfer, throughput optimization, TDMA, MAC, wireless sensor network applications, pipelining

1 Introduction

There have been several wireless sensor network deployments in the recent past. The modus operandi of most of these applications is to sense physical phenomena and communicate this data from individual sensor nodes to a common sink, typically over multiple wireless hops. We focus our attention on those applications which require bulk transfer of large amounts of sensed data. Notable applications of this kind are structural health monitoring [1, 2, 3], volcano activity monitoring [4], surveillance applications involving imaging/acoustics, etc. For instance, structural health monitoring applications collect vibration data at 200Hz [1, 2]. BriMon estimates about 7KBytes of data per sensor per train pass [3], and this data must be transferred to a sink as quickly as possible so that network nodes can sleep subsequently, to save power. Volcano monitoring reports as much as 50KBytes of 1 minute data per sensor [5]. In fact, [5] reports data transfer throughput as a bottleneck in the system design as the data transfer latency hampers node's ability to detect back-to-back (seismic) events. The above class of applications require reliable bulk data transfer and are throughput intensive. In battery-operated situations, higher throughput is important from the perspective of power savings too, since a node can sleep soon after transferring its collected data [3].

Accordingly, this paper considers the goal of achieving high throughput for reliable bulk data transfer. There are mainly three stumbling blocks that make this difficult in a multi-hop wireless setting. The first and the foremost is interference: inter-path, intra-path and external interference. Inter-path interference is that which happens between multiple flows, while intra-path interference is that which happens between multiple hops of the same flow. The second limitation is imposed by the hardware that is typically used in these deployments: low processing speed, limited RAM/hardware buffers. The third is the common culprit: wireless errors. We carefully design our solution to effectively handle each of these problems.

We first simplify the problem by considering only one flow at a time, thus eliminating the issue of inter-path interference much like in [5],[6]. Our framework easily extends to supporting two flows at a time; we shall show that this achieves close to optimal throughput, thus avoiding the need to support an arbitrary number of simultaneous flows which can come at considerable complexity.

SenSys'10, November 3–5, 2010, Zurich, Switzerland.
Copyright 2010 ACM 978-1-4503-0344-6/10/11 ...$10.00

We are thus faced with the challenge of extracting the maximum possible throughput along a path of nodes, with a single radio per-node. We have to achieve this while keeping the functionality simple enough to permit easy and robust implementation. Our solution is termed PIP, which stands for "Packets in Pipeline", and indicates that we seek to employ pipelining through spatial reuse for high throughput.

PIP uses a radically different approach from current solutions. Four features characterize the design of PIP. (1) PIP is part of the MAC module, and it provides a connection-oriented multi-hop MAC primitive, for use by the transport module. (2) PIP is TDMA-based, and nodes use synchronized time-slots. (3) PIP uses multiple channels for better spatial reuse and high throughput. (4) Finally, the above features of PIP are tied together with the use of central control. In the application space of our interest, the sink node is an ideal place for incorporating the centralized control.

While each of the above features themselves are by no means unique, we argue that the *combination* is unique to PIP and it is this *combination* which enables PIP to effectively address the earlier-mentioned three challenges, achieve high throughput, as well as have other useful features.

Since we consider one flow at a time, there is no inter-path interference. Features (2) and (3) eliminate intra-path interference. For our considered applications, external interference is not much of an issue (in remote locations like volcanoes, habitats, bridges, etc.), but we show how feature (3) can be adapted to do dynamic channel hopping to overcome external interference effectively. The centralized control coupled with modular design, make the protocol extremely simple that it can be implemented on limited capability hardware. Further, PIP is robust to wireless errors. Even when different links on the path have different error rates, PIP performs well without any dynamic flow control (unlike Flush [6]). And PIP can operate well with only moderate queue sizes.

We have implemented PIP on the Tmote-Sky platform, which uses the 802.15.4-compliant CC2420 radio chip. Our evaluation shows that PIP is effective in achieving high throughput; the throughput degrades only marginally with increasing number of hops.

Our prototype of PIP also includes a TinyOS-based implementation of the hardware pipelining suggested in [7]. The prototype is able to achieve about 60 Kbps throughput over a path of 10 nodes (9 hops). This represents over a twelve-fold improvement over the throughput achieved by current state-of-the-art [6] under similar conditions. We account for the various overheads in PIP and identify that PIP itself adds only minimal overhead.

The rest of the paper is organized as follows. The next section (Sec. 2) describes prior work and how PIP differs from these. Sec. 3 presents the detailed design of PIP. Subsequently, Sec. 4 presents an extensive evaluation of PIP using our prototype. Sec. 5 presents a few points of discussion and Sec. 6 concludes the paper.

		Centralized		Distributed	
		Single chnl.	Multi chnl.	Single chnl.	Multi chnl.
Uncoordinated (CSMA)	Connection less	NA	NA	***802.11*** ***802.15.4*** ***BMAC*** ***HOP***	DCC (two-radio)
	Connection oriented	NA	NA	***Flush***	NA
Coordinated (TDMA)	Connection less	NA	NA	***Overlay MAC***, ***FPS,*** TRAMA, LMAC, SMAC, TMAC	CHMA, CHAT, MMAC, MAP, SSCH, McMAC ***(1-hop)***
	Connection oriented	***RT-Link***	Wimax ***TSMP*** ***PIP***	NA	NA

Table 1. Summary of related MAC protocols

2 Related Work

Efficient data transfer in a multi-hop mesh network has received considerable attention. Below we compare our work with prior transport layer solutions, MAC layer solutions, and specifically with Flush [6].

Transport layer solutions: RMST [8] is a transport layer solution that outlines the architectural choices for bulk transfer. We employ some of the RMST's techniques in our work: such as hop-by-hop retransmissions and end-end SNACKs. These techniques are routinely applied for efficient data transfer (e.g. see Flush [6]). ESRT [9] is another transport layer protocol that seeks to achieve the required reliability at minimum congestion. The definition of reliability as applied to ESRT is very different from ours: ESRT seeks to transfer 'high percentage' of messages from a set of sensors to the sink, for accurate detection of events. PSFQ [10] is a transport protocol that looks at reliable programming of sensor nodes. PSFQ is basically a dissemination problem while ours is a collection problem.

In comparison to the above, our main contribution is careful design of a *cross-layer* framework that achieves maximal bulk transfer throughput. ∇

MAC layer solutions: Tab. 1 succinctly compares PIP with other MAC protocols [11, 12, 13, 14, 15, 16, 17, 18, 19]. As can be seen, PIP differs from most prior MAC in terms of its combination of four design choices: connection-oriented, TDMA-based, multi-channel, centralized. We wish to stress that although each of the four aspects by themselves are not unique, the *set* of design choices constitutes a non-obvious design point; this is what enables PIP to achieve high throughput. The fact that the combination of design choices is non-obvious is borne out clearly in Tab. 1: despite the variety in MAC layer solutions, only a few have chosen this combination for use in sensor network applications.

PIP is akin to TSMP [20] and Wimax [21] in its combination of design choices. While multi-hop Wimax is yet to be implemented or evaluated, TSMP has been. In comparison to TSMP, PIP's contribution is in terms of the design and optimization for bulk data transfer applications.

PIP is also unique in terms of its use of built-in time-synchronization, and its goal of high throughput bulk data transfer. Further, we have demonstrated the feasibility of our ideas via implementation. An implementation-based validation is of great significance since in any TDMA system, since the practicality of the time synchronization assumption is always in question unless shown otherwise. Tab. 1 shows in bold italics, the few protocols which have an implementation.

The time-sync in PIP is similar to that in TSMP [20] and FPS [22], in that we use simple per-hop synchronization rather than sophisticated beaconing (such as RBS/FTSP) for synchronization or drift computation. An important difference from TSMP & FPS though is the fact that we use the data packets themselves for synchronization along the path!

As pointed out in Table 1, there are several single channel as well as multi-channel MAC protocols in the literature (e.g. [19, 17, 22, 23]). But unlike PIP, none have been designed or optimized for bulk data transfer. (AppSleep [23] recognizes that a path of nodes can be made 'awake' during bulk transfer, but does not deal with bulk transfer itself). ∇

Flush: The closest that comes to our work is Flush [6], a cross-layer protocol that works under the same premise as ours i.e. one flow at a time. Flush uses a dynamic rate control algorithm that avoids intra-path interference. This bottleneck bandwidth is determined without any extra control packets but by snooping control information. The underlying MAC that flush employs is however CSMA.

The first major difference between Flush and our solution stems from the fact that we use time synchronized slots to clock out our packets. Here we leverage the well known fact that deterministic scheduling rather than stochastic scheduling is better for the saturated case [24]. The second difference stems from the fact that we use multiple channels to increase spatial reuse. We note that the very design of Flush cannot support multi-channel operation since the design depends on snooping to decide when to send packets. Apart from this, a third difference is that we also employ a hardware-specific optimization as suggested in [7]. This optimization moves packet copying off the critical path, and is specific to the hardware platform. This optimization does not apply to Flush since packet copying is not the bottleneck in Flush.

These three differences put together permit us to achieve significant performance improvement over Flush: a factor of over twelve (1200%) for a path of ten nodes. Furthermore, we evaluate PIP's robustness under varying channel error rates. ∇

802.11 TDMA implementations: In another domain, namely 802.11, there has been recent work on TDMA implementations: [25, 26]. Most of the measurements in [25] are on a single hop, while [26] too evaluates only two hops at most. The work in both [25, 26] has focused on time synchronization. In contrast, PIP is a cross-layer design to achieve high throughput, and its design has been validated over 802.15.4, an impoverished radio, over up to 9 hops.

3 The Design of PIP

We now present the design of PIP. In the explanation below, we consider only a single flow at a time. At the end of the section, we show how PIP can be extended to support two flows at a time and how this achieves close to optimal throughput.

3.1 Design choices

There are four important choices we made in PIP's design.

(1) TDMA versus CSMA: While CSMA allows for an easy distributed implementation, it is known to suffer from poor performance, especially for bulk data transfer, due to intra-path interference. PIP chooses a TDMA-based approach. The common knowledge about TDMA is that time synchronization is a difficult practical issue. PIP breaks this myth and shows that it is possible to clock packets efficiently using a simple *piggy-backed* time-synchronization mechanism. ∇

(2) Multi-channel versus single-channel: A TDMA-based approach also allows us to use multiple channels for further throughput enhancement. If we use a technology like 802.15.4, as we have in our prototype, we have several channels: 802.15.4 has 16 different channels in the 2.4GHz ISM band (although 4-5 are sufficient to achieve spatial reuse). The same slotting structure used by the TDMA mechanism is also used as the granularity of any channel switching. Channel switching does add a small but noticeable overhead, but the benefits of multi-channel operation far outweigh this, as we show in our prototype implementation. ∇

(3) **Centralized versus distributed:** While distributed protocols can potentially be simple, seeking to optimize performance generally introduces complexity. For instance, distributed TDMA or distributed multi-channel protocols are necessarily more complex than their single-channel counterparts. Centralized approaches are generally frowned upon for two reasons: lack of fault tolerance, and lack of scalability. However, as mentioned in [27], most sensor network deployments have a central sink node anyway, which collects the sensed data. The central sink in most cases also has better resources (battery, memory, CPU, etc.). This becomes a natural point of central coordination. We thus choose a centralized approach for the PIP MAC. ∇

(4) Connection-oriented versus connection-less: A connection-oriented MAC means that a higher layer can use it only after a connection formation phase. Most MAC protocols are connection-less, with the notable exceptions of Bluetooth and WiMAX [21], although the implementations of both these wireless standards have only a single-hop connection formation. In PIP, the MAC layer provides a multi-hop connection-oriented primitive to the transport layer. This fits in well with our choice of using a TDMA-based multi-channel approach. With central control, the connection formation phase is used to specify the time-slot and channel of operation of each node in a data flow path. ∇

The above combination of design choices drives the design details below. The design below involves details as well as several subtle aspects, which we point out as we describe.

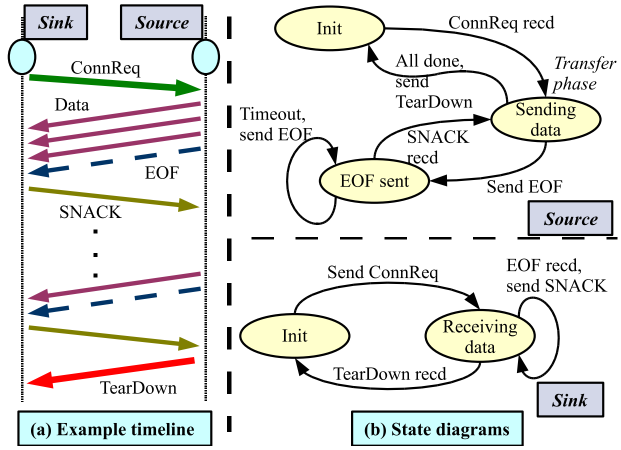

Figure 1. Transport protocol: timing and state diagrams

3.2 Transport protocol

In a system using the PIP MAC module, the transport protocol has one simple role: to handle end-to-end reliability. There are however some subtleties and implications for the PIP MAC, which we point out below.

The operation of the transport protocol is shown in Fig. 1(a). The transport protocol itself is connection oriented, and starts with a connection *setup* phase. Subsequently we have one or more data *transfer* phases. Each transfer phase ends with the source explicitly requesting a SNACK (Selective Negative Acknowledgment), by sending an EOF (End-of-File) packet[1]. Depending on which data packets were (negatively) acknowledged by the sink, the source retransmits packets end-to-end. This cycle repeats as often as is necessary, and finally, the source sends a TearDown which closes the connection.

There are three subtle points to note. First, the connection request originates from the data sink. Second, the source explicitly requests a SNACK packet by sending an EOF. That is, the sink does not send a SNACK packet on its own. Third, the TearDown originates from the source. Although the transport protocol could have been designed differently with respect to these three aspects, we have made the above choices for specific reasons, which we shall explain when we describe the PIP MAC protocol (Sec. 3.4).

Fig. 1(b) shows the transport protocol state diagram at the source as well as the sink. There are exactly two timers in the transport protocol: one at the source node and one at the sink. The source node timer is started after sending an EOF packet, expecting a SNACK packet. So this timer handles EOF packet losses as well as SNACK packet losses. The sink node timer is used to detect loss of a ConnReq packet. When the timer goes off, the transport module may decide to either report failure of connection establishment to the application module. Or it may resend the ConnReq packet.

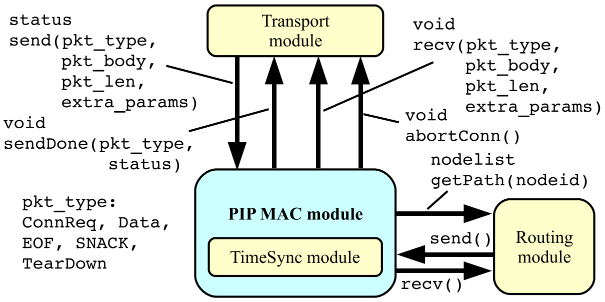

Figure 2. PIP MAC, transport, routing, time-sync modules

3.3 PIP: Functionalities and Interfaces

We now discuss how PIP interfaces with other modules. Specifically, we discuss four functionalities: PIP MAC, time-synchronization, routing, and transport. The cross-layer design and the interfacing between the four functionalities is best illustrated by means of a diagram: Fig. 2.

In our system, the transport module interacts directly with the PIP MAC module. And the PIP MAC module provides a multi-hop, connection-oriented primitive for use by the transport module. At the same time, the PIP MAC module provides a separate connection-less, single-hop interface for use by the routing module. The connection-less interface is based on regular CSMA/CA. The routing module may use it to send any message, depending on the routing protocol used.

Our MAC module thus provides two different kinds of interfaces, for the transport and routing modules respectively. It also encompasses a time synchronization module within itself. We do not rely on any additional time synchronization mechanism, than what is possible using the data flow itself. The time synchronization module is active only during an ongoing data transfer. We explain this further when we discuss the MAC protocol (Sec. 3.4).

The interface between the transport module and the PIP MAC module is depicted in Fig. 2. It consists of four calls: one of these is a command from the transport module, and the remaining three are signals (or events) from the PIP MAC module to the transport module. The calls are parameterized with the packet type. The possible packet types are exactly the same as the five used by the transport protocol: ConnReq, Data, EOF, SNACK, TearDown.

Thus the transport module (sink) initiates a PIP MAC connection by calling send with the ConnReq packet type. At the other end (the source), the PIP module signals a recv with the ConnReq packet type. The exchange of Data, EOF, SNACK, and TearDown packets happens similarly.

The PIP module may abort an existing connection by using the abort signal. Note that this is different from a normal termination using a recv with the TearDown packet type.

As part of its interface to the transport module, PIP provides only unidirectional data transfer at a given time. Some details of this interface are not shown in Fig. 2. First, the transport module conveys the data source for the connection as part of the connection request, as part of the extra param-

[1] Although we are calling this the EOF packet, the transport protocol as well the PIP MAC would work just as fine if the source sent an EOF in the middle of its data transfer, to trigger a SNACK.

eters in the send(ConnReq, ...) call. Second, the transport module gets information about the number of hops in the path, from the PIP module (which in turn gets this information from the routing module). This hop information is used to set the transport timer values. Third, as part of the PIP-transport interface, PIP guarantees that packets will not be delivered out of order.

The interface between the routing module and the PIP MAC module is depicted in Fig. 2. Apart from the usual send and recv calls, we have a getPath routine which is used by PIP MAC only at the central sink node. This routine returns the sequence of nodes in the path from the sink to the given source node in the network. This information is used by PIP in determining the schedule and in turn establishing the multi-hop MAC connection to/from the source node.

Here we have assumed that the routing module at the central node is indeed capable of returning the path to/from a given (source) node. This is possible for example, with a link state or a path vector style routing algorithm, or even with centrally controlled routing protocols like the one in BriMon [3]. The operation of PIP is dependent on the routing protocol to this extent, but is otherwise independent of it.

3.4 PIP MAC protocol

The PIP MAC module provides three main functionalities: (1) the connection-oriented interface to the transport module, (2) the embedded time-synchronization mechanism, and (3) the connection-less interface to the routing module.

The three modes of operation: In implementing the connection-less and the connection-oriented interfaces, PIP switches between three overall modes of operation. (1) An unsynchronized, single-channel mode, abbreviated as *U1C*, is used while providing the connection-less interface. (2) A time-synchronized, multi-channel mode, abbreviated as *SMC*, is used while providing the connection-oriented interface. In addition, while transitioning from *U1C* to *SMC*, PIP uses an intermediate unsynchronized multi-channel mode *UMC*.

The *U1C* mode is used when there is no ongoing flow. By design, all the nodes in the network will be in the same channel. In fact, we reserve one of the radio channels for the *U1C* mode, and call it the *U1C* channel. This channel is not used in the *SMC* or *UMC* modes in any of the nodes in the network. The *SMC* mode is used during data transfer, for high throughput. Notably, *SMC* is used only during data transfer. We now outline PIP's operation to effect a data transfer. ∇

Connection setup phase: PIP recognizes the same five kinds of packets as is used by the transport module: ConnReq, Data, EOF, SNACK, and TearDown. Fig. 3 shows the PIP MAC protocol in operation across a path of five nodes. The transport module at the sink E starts proceedings by making a connection request to the PIP module. This request specifies the data source as A. The PIP module at E then queries the routing module at E for the path to the data source A.

The central sink node then decides two important things. First, for every node in the path, it assigns a designated *receiving channel*, for use in the *UMC* and *SMC* modes. We also term this the *SMC* channel of a node. In the example, $A \ldots E$ are assigned *SMC* channels 12...16 respectively (11 is the *U1C* channel). If a node X has to transmit any packet to its neighbour Y, it switches to Y's *SMC* (or receiving) channel and transmits.

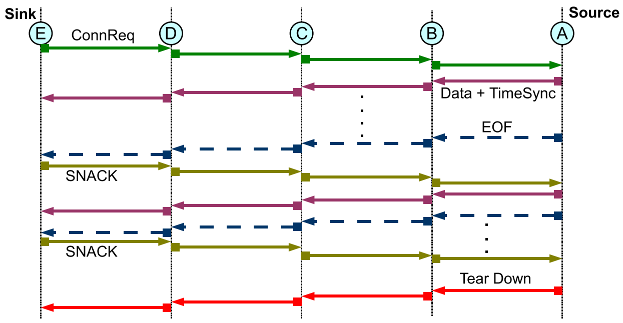

Figure 3. PIP MAC operation: an example

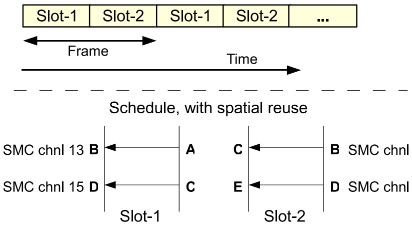

Figure 4. Illustrating frame, schedule

The second important thing decided by the sink node is the *schedule*. While describing this, we need some terminology related to TDMA systems. For this paper, we shall term a *slot* as a time span which represents the unit of reception (or transmission, by a neighbouring node). A *frame* is a sequence of slots; this same sequence of slots repeats over time. A *schedule* is a specification of which nodes will receive in which of the slots in the frame.

The *schedule* as well as the *SMC* channel for each node is determined centrally, by the sink node. As an example, the *schedule* and *SMC* channel for each of the nodes in Fig. 3 is shown in Fig. 4. The sink node then conveys this information in the PIP header of the ConnReq packet, as this packet is forwarded toward A.

Note that the algorithm for computation of the schedule and the set of *SMC* channels is itself independent of PIP's operation. We have used a particular simple yet efficient algorithm for this, but any other mechanism can be used as well.

An important point to note is that the MAC modules at the intermediate nodes along the path maintain state about the connection. Specifically, each node learns the time-slot and channel information from the ConnReq packet as they forward it along. Each node also learns its neighbouring nodes and their time-slot and *SMC* channel information. These details are used during the data transfer.

After forwarding the ConnReq, a node changes from *U1C* mode to *UMC* mode. Once the ConnReq reaches the PIP module at A, the module strips off the PIP header and gives the packet to the transport module. The transport module then uses the transport level information in the request to decide which data to transmit. It then starts to transmit Data packets. At this juncture it is useful to discuss our time synchronization scheme. ∇

Time synchronization: Now, we have the *UMC* mode for a specific reason. Although the sink is the one which centrally computes the schedule, this schedule is only logical (i.e. with respect to a certain starting time). The actual clocking for the schedule is provided by the data source. That is, every Data packet carries embedded time synchronization information (Fig. 3).

Every data packet uses a timestamp field (4 bytes) which is used to synchronize neighbouring nodes. In implementation, the timestamp is inserted by an interrupt triggered by the Start of Frame Delimiter (SFD). This is the same mechanism used in [3, 28]. Thus, on reception of the first Data packet, a node changes from *UMC* mode to *SMC* mode. An exception to this rule is the source node, which skips the *UMC* mode and enters *SMC* mode directly on reception of a ConnReq.

Note that no transmission can happen in *UMC* mode, since the node does not yet have clocking information. As a Data packet is received and forwarded, a node embeds (its knowledge of) the global clock so that its downstream neighbour can synchronize with it. And as data packets are handed from the source's transport module to the source's PIP module, they keep flowing from *A* to *E*, and nodes maintain synchronization.

Thus the data source node is also the source of the clock for the entire path. Note specifically that, unlike [29, 30], we do not have any clock drift estimation. This is a conscious choice; our mechanism is sufficient and efficient, as we show in Sec. 4.4. ∇

EOF, SNACK exchange: Like the Data packets, the EOF and SNACK packets are also forwarded faithfully by PIP, with all nodes operating in *SMC* mode. PIP passes these packets onto the transport layer only at the source or the sink, not in the intermediate nodes.

There is a subtle aspect in the specification of the schedule and the *SMC* channels. These specifications are based on *reception* rather than node *transmission*. This is so that the same schedule and *SMC* channel information can be used easily for the forward direction (Data, EOF, and TearDown) packets as well as the reverse direction (SNACK) packets. This however does not introduce any "collision" at any node (where transmissions can come from either neighbour at the same time). This is because, by design, we have packets only in one direction at a time. This is the case unless we have scenarios of premature SNACK timeout at the source, which we expect to be rare.

The above reason also explains why the data sink does not send SNACK packets on its own, and only when triggered by an EOF from the source. After it sends an EOF, the transport module at the source is expected to *wait* for the receipt of a SNACK packet; specifically, it does not continue sending data packets. In this scheme, the SNACK packet faces no contention from packets in the opposite direction. The EOF clears out the Data packet queue at intermediate nodes, and hence the SNACK faces no packets in the opposite direction when going from sink to source.

Data and EOF packets carry synchronization information. But SNACK packets, since they travel in the opposite direction, do not. So for a period of time, after the EOF is forwarded and before getting the next Data packet, every node does not get updated synchronization information. This can potentially lead to a loss of synchrony due to clock drift. However, as our measurements show, the maximum drift is small, and the drift in one round trip time is not significant. ∇

Connection tear-down: When the source's transport module determines that all data packets have been acknowledged, it initiates connection termination by using a TearDown. As the TearDown travels from *A* to *E*, the PIP modules in the intermediate nodes change themselves from *SMC* mode to *U1C* mode. The connection is deemed to have ended. ∇

It appears on the surface that there is no reason for the EOF packet to be recognized specifically by the PIP MAC modules. However, there is a subtle reason for this. The PIP MAC modules maintain the queues at all of the network nodes. They treat EOF as a special packet, and do not allow it to be dropped due to queue overflow. This reduces the chances of a transport module timeout waiting for a SNACK packet.

In our design, we initially considered the possibility of having the TearDown sent by the sink node itself, instead of sending the final SNACK acknowledging all packets. But such an approach does not handle TearDown packet losses well, due to the following reason. Recall that the PIP module changes from *SMC* to *U1C* mode after forwarding a TearDown packet. An inconsistent state can result, as follows. Say, when *C* is sending a TearDown to *B*, all the transmissions get lost. Now, no end-to-end retransmissions are possible since *E* and *D* have already changed to *U1C* mode and are in their *U1C* channel. So we now have the sink thinking that the transfer is complete, while the source thinks otherwise. To avoid this inconsistency, we have the TearDown packet sent by the source node, toward the sink. In such a scenario, even if all the retransmissions of the TearDown packet get lost on a particular hop, it is alright. The transport modules at both the source and the sink are aware that all data packets have indeed been received. ∇

Handling wireless errors, hop-by-hop ACKs: For all packets, the PIP module implements hop-by-hop ACKs for efficiency. The time-slots are long enough to accommodate an ACK packet. There is a maximum number of retries after which the PIP module gives up retransmission of a particular packet. An important point to note is that retransmissions happen only during the next transmission opportunity of that node (as defined by the time-slot schedule). ∇

PIP state diagram, timers: Fig. 5 shows the PIP MAC state diagram. There are three states shown on the top row, and two on the bottom row. The top three states correspond to *U1C* mode. And the bottom two states, drawn with thick border, correspond to *SMC* mode (after reception of the first data packet).

In PIP, a ConnReq packet is sent using the *U1C* channel in an uncoordinated fashion. And a loss of the ConnReq

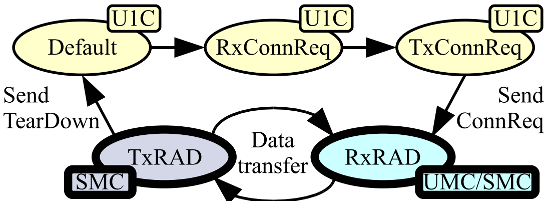

Figure 5. PIP MAC state diagram

packet leads to a connection failure at the transport module. To avoid this, three copies of the ConnReq packet are sent (an intermediate node may receive only one copy, but still it will forward three copies).

The RxRad and TxRad are the coordinated data transfer states in the *SMC* channels, as per the schedule received from the sink node in the ConnReq packet.

Some state transitions are not shown in Fig. 5, for clarity. For instance, the sink node skips the RxConnReq state, while the source node skips the TxConnReq state. Similarly, the sink node transitions directly from RxRad to Default on receipt of a TearDown. Also, timeout-based state transitions are not shown in the state diagram.

The PIP MAC module has only two timers, and both are for entering the *U1C* mode from the *UMC* or *SMC* modes. These are safe-guards against unexpected and rare situations such as node failures or repeated wireless errors (beyond what can be handled by the hop-by-hop and end-to-end retransmissions).

One of the two timers waits on data packets from the source to the sink, and another on hop-by-hop ACKs from the next-hop toward the sink. For instance, C has a timeout for expecting packets from B and another timeout expecting ACKs from D. If either of these timers fire, C aborts the ongoing connection and enters *U1C* mode. If the timeout had been caused due to, say, failure of B or D, the routing mechanism can then kick-in for appropriate repair. This then allows the possibility of the transport module, or the application module, issuing a fresh connection request after the repair.

In summary, though there are many approaches possible in the identified design space, considerable attention was paid in PIP to keep the underlying modules simple, yet robust to permit easy implementation. ∇

3.5 PIP Prototype Implementation

We have implemented a prototype of PIP using the Tmote-Sky hardware platform, which is based on the 802.15.4-compliant CC2420 radio. The software platform we have used is TinyOS.

SPI bottleneck and pipelining

In the platform we used (Tmotes), there happens to be a system bottleneck in data transfer, as identified in [7].

The Tmote-Sky platform has a SPI (Serial Peripheral Interface) between the MCU (Micro Controller Unit) and the radio. Each packet has to be copied from/to the radio to/from the MCU at each forwarding node. In such a case, the SPI becomes the bottleneck. This problem is identified and solved in [7]. The solution is to parallelize radio and SPI operations at intermediate nodes. Packet i's transfer from the radio to the MCU is done in parallel with packet $i-1$'s radio transmission. And, packet i's transfer from MCU to the radio is done in parallel with packet $i+1$'s radio reception. This is shown in Fig. 6.

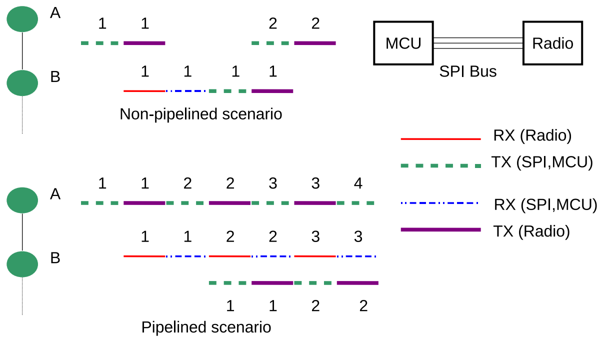

Figure 6. Pipelining SPI and radio opns. in the Tmote-Sky

It is evident from the diagram that the throughput is doubled as compared to normal transfer throughput. Note that this optimization is possible because the speeds over SPI and radio are comparable (in fact, the SPI is slightly faster than the radio). We have implemented this pipelining optimization in TinyOS.

Time-slot and channel allocation

In 802.15.4, we have 16 channels (numbered 11 to 26) in the 2.4GHz band[2]. In our implementation, we choose channel 11 to be the *U1C* channel. The *SMC* channel for a node is determined by its hop count from the sink. The sink uses channel 12 as its non-default reception channel, the next node is assigned channel 13, and so on.

For the time-slot schedule, we simply have a two-slot schedule in implementation: slot-1 and slot-2. The source node, and all nodes at even hop-count from it, have slot-1 as their reception slot (and slot-2 as their transmission slot). And all nodes at odd hop-count from the source have slot-2 as their reception slot (and slot-1 as their transmission slot). Such a simple scheme works so long as there is no interference for a node beyond k hops, where k is the number of data channels in use. In our experience, $k=4$ or $k=5$ usually suffices to avoid intra-path interference, but this could depend on the environment; in any case we have as many as 15 data channels with 802.15.4 in 2.4GHz. The *SMC* channel allocation and time-slot scheduling in our implementation are illustrated in Fig. 4.

Time synchronization requirement

In the above two-slot schedule with multiple channels, we now explain how PIP's operation needs to worry only about the synchronization error between neighbouring nodes. Fig. 7 shows the global clock, and also the slot/frame boundaries as interpreted by different nodes. These slot/frame boundaries are different from the ideal slot/frame boundary due to synchronization errors. The interpretation of guard time is: (a) a node does not transmit

[2] Channels 1-10 are not in the 2.4GHz band [11].

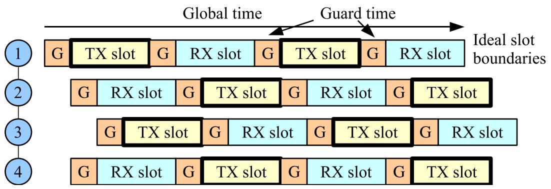

Figure 7. Illustrating PIP's time sync. requirement

into the guard time, as per its notion of slot boundaries, and (b) a node potentially continues reception into the guard time following its RX slot

Suppose we assume that a channel is reused only after so many hops that the reusing node is anyway out of interference range, as explained earlier. It is then easy to see from the diagram that as long as neighbouring nodes' synchronization errors are accounted for in the guard time calculation, we do not really need to worry about synchronization errors adding up with increasing number of hops[3].

3.6 PIP and Throughput Optimality

50% of optimal throughput: We start with the claim that the throughput achieved in this system is close to optimal as long as the sink is kept continuously busy. We justify this claim in Sec. 5 after presenting the performance results.

So far, we have considered only one flow active at a time. In the data path of this flow, all nodes except the source and the sink are busy all the time (either transmitting or receiving a packet in each slot). However, the sink is busy only half the time since it is only receiving. So in a system where all data has to be finally transferred to the sink, we have so far achieved close to 50% of the optimal throughput.

Extending PIP to achieve optimal throughput: It is conceivable that the sink can be involved in two different connections simultaneously, thus fully utilizing its radio. In such a setting, we would achieve optimal throughput. There are two requirements for this. (1) First, all nodes must be synchronized to the sink, instead of to the source (as was the case for the single flow). (2) Further, we also need to ensure that the two flows do not cause inter-path interference.

The first requirement is easy to achieve. PIP can provide a periodic back-channel to convey synchronization information. As we shall see in our evaluation, the clock drift is a negligible component of synchronization error. So if the back-channel can be as infrequent as once in several hundreds of frames.

The second requirement, that of ensuring that the data transmissions of the two flows do not interfere with each other, would ideally need an interference map of the topology to do appropriate channel allocation. But constructing and maintaining such a map comes at considerable complexity. A simple alternative, if sufficient channels are available, would be to use a disjoint set of channels for both the flows. As discussed earlier, if we assume that $k = 5$ channels per flow is enough to provide sufficient spatial reuse, thus requiring at most 11 independent channels ($2 * 5$ + 1 control channel) for two flows. This is feasible in 802.15.4 in 2.4GHz since it has 16 independent channels.

4 PIP: Performance Evaluation

While most of our evaluation is prototype-based, for comparison, we have also used a Markov-model based theoretical analysis, as well as a simulator. We describe the setup for these here, and subsequently present the prototype performance results. The comparison of the prototype results with analysis and simulation convinces us that the TDMA implementation is indeed robust.

4.1 PIP: Markov Analysis

It is interesting and always useful to evaluate the merit of a protocol through appropriate theoretical analysis. In PIP, we model the buffer occupancy at each node as a discrete time Markov chain. The analysis assumes that the input process to each node is i.i.d Bernoulli process. The output from one node, e.g. throughput value, is the input to the next node in the path. We omit the details of Markov-chain analysis due to lack of space. The analysis gives us (1) the utilization (throughput) of the system for given number of hops, queue size limit and error rate (2) the probability of queue drops for given queue size limit and error rate (3) the probability of slot being idle. In particular, we use the analysis to set the queue size limit considering tolerable probability of queue drops for a given error rate. The analysis also gives us the theoretical upper bounds to compare and verify the throughput achieved in our prototype implementation.

4.2 Simulation-based evaluation setup

We have built our own hand-crafted simulator to quickly evaluate the performance of PIP through a range of parameters and design choices. Specifically, we make use of the simulator to study the (lack of) need for flow control in PIP. The simulator captures all the relevant design aspects of PIP. In comparison to the analysis, the simulator does not approximate the input process of intermediate nodes as being Bernoulli i.i.d. It also implements a maximum retry limit for hop-by-hop retransmissions, not modeled in our analysis.

4.3 Prototype implementation setup

We evaluate the performance of our prototype on a 10-node, 9-hop setup, unless mentioned otherwise. Our prototype includes an integration with the C2P routing protocol [3]. However, for the evaluation results below, we have shunted out routing in the evaluation (i.e. each node has a pre-determined forwarding table). This gives us better control over the number of hops, and the presence/absence of routing does not affect the data transfer throughput anyway. The channel and schedule allocation are as explained in Sec. 3.5.

For most of the experiments presented, all the nodes are placed on a table near one another, for ease of setup. This does not matter since PIP uses different channels for adjacent links. Indeed, we have verified that we get similar behaviour even when the nodes are several metres away from one another.

To carefully study PIP's behaviour under various loss conditions, we optionally *emulate* wireless channel losses. We do this by making a node, probabilistically drop packets (original or retransmitted) that are ready for transmission.

[3] It is worth observing that in a general setting, what matters is the synchronization error between nodes $N+1$ hops away, where N is the interference range.

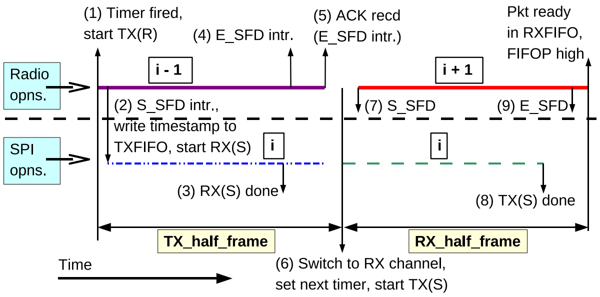

Figure 8. Slot operations in the PIP prototype

For the various experiments based on the implementation, we transfer 1000 packets overall. The MAC payload is set to 103 bytes, to which 12 bytes of 802.15.4 header, 3 bytes of PIP header, 4 bytes of time-stamp (tailer) and 2 bytes of CRC are added. This gives us an overall packet size of 124 bytes and leaves space for 4 more bytes in the CC2420 RX-FIFO (whose size is 128 bytes), which is necessary for accommodating the hardware ACK. Thus we make full use of the available buffer space in the radio.

In presenting many our results below, we use the Tmote-Sky clock *tick* as one of the units of time. $1tick = 1/32KHz \simeq 30.5\mu s$.

4.4 Slot duration and guard time

The performance of a TDMA system is related to two important parameters: the slot duration and the guard time. In the PIP prototype, we have used a slot duration which is sufficient to accommodate one full size packet (128 bytes, including ACK, in the CC2420). And we use a guard time as small as possible, for maximum efficiency. We now determine what PIP's slot duration and guard time should be.

Fig. 8 shows the various operations which happen in our implementation, in the two slots of our schedule. The figure shows radio operations with a (R) and SPI operations with a (S). The radio operations are shown on top, while the SPI operations are shown at the bottom. The sequence numbers $i-1, i, i+1$ correspond to different packet numbers. As explained in Sec. 3.5, packet i's RX(S) is done in parallel with packet $i-1$'s TX(R). And in the very next slot, packet i's TX(S) is done in parallel with packet $i+1$'s RX(R). In the figure, the S_SFD and E_SFD events correspond to the start of frame delimiter and end of frame delimiter, both of which are delivered using the same interrupt called SFD.

Clearly, we can see that the slot duration is defined by the maximum of the four operations: transmission over radio, reception over radio, transmission over SPI, and reception over SPI. In our implementation, with the use of DMA, the SPI operations are significantly faster than the radio operations. Our slot duration is determined by the transmission duration and related processing overheads, which we account below.

In our implementation, we have measured the time between start of the timer for the start of the transmission slot, and the start of switching to the receiving *SMC* channel. That is, events (1) and (6) respectively in Fig. 8. This time duration is about 200 ticks. In this, about 175 ticks are spent until the time we receive the ACK's E_SFD interrupt indicating that the ACK has been fully received.

Component	Measured value
Inaccuracy in timer fire	1-2 ticks
Time-sync error	mostly under 1 tick
Processing jitter	1-2 ticks
Channel switching time	$\simeq$ 10 ticks

Table 2. Components of guard time

Of the 175 ticks, about 138 ticks corresponds to the on-air time of the 128-byte data packet and the 4-byte ACK packet. And the CC2420 chip specifies that the ACK transmission will begin 12 symbols (6 bytes) after reception of the data packet. This adds a further 6 ticks which we can account for. We attribute the rest of the time to various software overheads such as interrupts; our processor is only an 8 MHz processor. Our current implementation also includes various logging statements and sanity checks included for easy testing and debugging.

Accordingly, we set the slot duration to be 200 ticks. This is about 1.5 times the ideal value of just the 128-byte transmission time, which is 134 ticks.

In any TDMA system, we need a guard time for each slot. In PIP, we define *guard* time as the duration to be given as leeway prior to each slot. There are several components which contribute to the guard time. (1) Inaccuracy in timer fire: we may set a timer to fire at time t, but it fires at $t \pm \delta$. (2) Time synchronization error between neighbouring nodes (as explained in Sec. 3.5). (3) Processing jitter: we expect an operation to complete within a certain time, but it could take longer. (4) Finally, the channel switching time.

Table. 2 summarizes the above four values, as measured on our prototype. We accordingly set the guard time to be 15 ticks.

In PIP, between the time a node forwards an EOF packet, and the time it receives the next Data packet, it takes one round trip time, which is proportional to the number of hops[4]. During this time, there is no time-sync correction available and the node's clock may drift. However, this drift is negligible: only about 20-40 μs per second [3, 29], or about 1-2 ticks per second. Hence any such drift too is accommodated by the above guard time easily.

A frame consists of two slots, along with their corresponding guard times. Thus the frame duration is $(200+15) \times 2 = 430$ ticks.

4.5 Queue size requirement

Another important parameter that influences PIP's performance is the queue size limit at a node. The purpose of a queue in an intermediate node in PIP is to account for *temporary* variations in error rates of the incoming versus the outgoing wireless hop. We stress "temporary", since any long-term imbalance makes the queue useless. If in the long-term, the incoming error probability exceeds the outgoing

[4] On a path, the latency for a single packet is 3 slot durations, or 1.5 frame durations, per hop. This is because, of the four phase: RX(R), RX(S), TX(S), and TX(R), only one overlaps with the four phases of the neighbouring node.

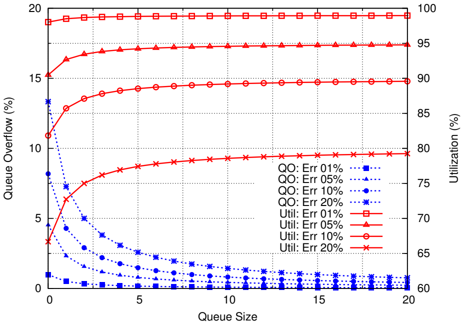

Figure 9. Throughput vs. queue size: various error rates

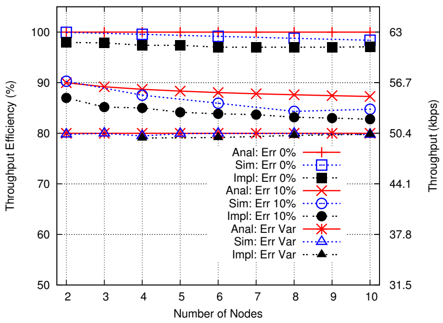

Figure 10. Throughput vs. NumNodes, various error rates

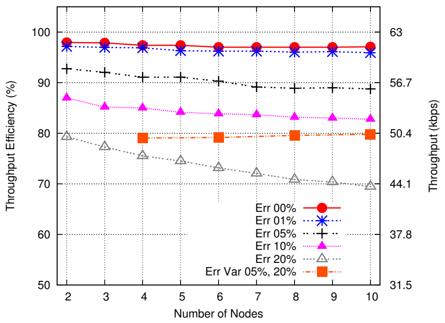

Figure 11. Throughput in PIP prototype implementation

error probability, there will not be any queue build-up. Vice-versa, if the error probability on the outgoing link is higher in the long-term, then even a large queue size will result in consistent losses.

Hence we consider an analysis of the situation where the wireless error rates of all the links are uniform since this is when queue size limit matters most. A point worth noting here is that, in such a path, the performance of the *first* intermediate node is always the worst. This is intuitive since the output rate of the source is 100%, whereas the output rate of any intermediate node is less than 100%.

Fig. 9 shows the performance of system calculated from our Markov analysis from the perspective of the *first* intermediate node. The left-side y-axis shows the probability of queue drops, while the right-side y-axis shows the utilization of the outgoing link. We see that even for error rates as high as 20%, beyond a queue size of 10, the performance flattens out. We thus use a queue size of 10 in our implementation.

4.6 Comparison of analysis, simulation, and implementation

Fig. 10 plots the throughput achieved in our prototype, as a function of the number of nodes in the path, for different error rates. We show the plots for 0% error rate[5], 10% error rate, as well as a variable error rate scenario. In the variable error rate scenario, all the links of the path have 5% error rate, while the last hop has a 20% error rate. For comparison, Fig. 10 also plots the corresponding numbers from analysis as well as simulation.

The y-axis (left and right) shows both the throughput efficiency and the actual throughput in Kbps. We measure efficiency as follows. The source transmits a packet in every frame (which consists of two slots). Thus ideally we should receive one packet per frame duration at the receiver. Efficiency is defined as the ratio of the number of packets received at the sink to the number of frame durations. For the implementation, only the *transfer* phase throughput is shown: discounting the connection-setup, SNACK round-trip, and tear-down phases. The throughput as shown on the right y-axis is computed for a 103-byte payload supported by PIP for the transport module (Sec. 4.3), and a 430-tick frame duration (Sec. 4.4).

We note that the results from the analysis, simulation, and implementation match very well. And all of these match closely with what we expect ideally: close to 100% utilization for 0% error rate, and close to 90% utilization for 10% error rate. We also observe that the utilization goes down only slightly with increasing number of hops.

The match between analysis, simulation, and implementation is the closest for the variable error rate scenario, where the last hop is the bottleneck bandwidth with 20% error rate. The reason for this is that in this scenario, the losses are dominated by queue drops at the penultimate node. In other words, the system behaves as if there is a single link with 20% error rate.

4.7 Throughput as a function of error rate

We further present the throughput performance of our prototype implementation. Fig. 11 shows the throughput from our prototype as a function of the number of nodes in the path. Node-1 is the source of data packets. The different plots show the performance at various (emulated) error rates. We also show the variable error rate scenario from earlier, where all the links of the path have 5% error rate, except the last hop which has a 20% error rate.

We make various observations. The throughput drops only slightly with increasing number of hops; only the 20% error rate case shows a noticeable drop. Also, the throughput in each case matches closely with what we expect ideally for that error rate. That is, efficiency loss is almost always due to the underlying wireless error rate, not due to the protocol.

PIP's throughput of about 60 Kbps for a 10-node (9-hop) case represents more than a twelve-fold improvement over the 6-hop throughput of about 600 bytes/sec reported by Flush (Fig. 12 in [6]), under similar test conditions.

Now, 802.15.4 has a PHY data rate of 250 Kbps. Ideally, a single radio per node system should be able to achieve 125 Kbps (since a radio of an intermediate node cannot both transmit and receive simultaneously). So PIP's throughput of 60 Kbps is close to half this idea value. The above calculation discounts the header overhead. If we factor in the headers (total bits on air) the throughput achieved is about 78 Kbps, which is about 2/3rd of the maximum of 125 Kbps. We believe that it is possible to further optimize various aspects of our implementation by paying careful attention to interrupt processing. This in turn can cut down on the slot duration, thereby giving even better throughput.

[5] In implementation, when we say 0% error rate, this means that we did not emulate any wireless losses; of course we cannot control any wireless losses which do happen.

Event	Time in Ticks (Frame Number)
ConnReq sent by sink	0 (0)
ConnReq received by source	8756 (21)
First **EOF** sent by source	438626 (1020)
First **SNACK** received by source	457890 (1065)
Second **EOF** sent by source	569563 (1325)
Second **SNACK** received by source	584382 (1359)
TearDown sent by source	584687 (1360)
TearDown received by sink	589734 (1372)

Table 3. Time-line of a complete transfer

No. of Hops	No. of Nodes	**Fetch**			**Flush**			**PIP**		
		Single-flow thr' put (Kbps)	Single-flow Latency (sec)	Total Latency (sec)	Single-flow thr' put (Kbps)	Single-flow Latency (sec)	Total Latency (sec)	Single-flow thr' put (Kbps)	Single-flow Latency (sec)	Total Latency (sec)
1	1	4.4	94	94	11	38.35	38.35	59.4	7	7
2	6	2.6	157	942	9.4	44.88	269.29	59.2	7.1	35.6
3	5	1.9	220	1100	7	60.27	301.35	58.6	7.2	36
4	2	1.4	283	566	6.2	68.05	136.09	58.5	7.2	14.4
5	1	1.2	346	346	5.4	78.13	78.13	58.5	7.2	7.2
6	1	1	409	409	5	84.38	84.38	58.4	7.3	7.3
				3457			**907.59**			**107.5**

Table 4. Comparison of transport protocols

4.8 Details of an experimental run

We now present statistics from a complete run of PIP for a 10-node path, with 10% emulated wireless loss rate on each link. Table 3 shows the time-line for such a run. As we can see, there are two transfer phases, in addition to the connection setup and tear-down phases.

The overall throughput, including the connection setup time, SNACK exchange time, and tear-down, is thus $(1000 \times 103 \times 8)/(589734 \times 30.5) = 45.8 Kbps$.

For a similar experimental run, the packet drop statistics at each of the intermediate nodes is shown in Fig. 12. This is shown as a percentage of the 1000 packets sent from the source. We see that queue drops as well as drops due to maximum retry limit being exceeded, are minimal.

For a specific experimental run, a snapshot[6] of the queue length variation is shown in Fig. 13, for nodes 2 and 9, which are one hop away from the source and sink respectively.

4.9 External interference and channel hopping

PIP's design directly takes care of inter and intra-path interference but what about external interference? Our envisioned usage is in environments that will likely not have external interference: on top of volcanoes, remote habitats, free-way bridges etc. However it is interesting to evaluate how TDMA-based PIP performs in the presence of external interference, if any. Thus to tolerate and mitigate the level of interference, we modify PIP to support frequency hopping (much like how Bluetooth does) as follows. Once data transmission begins, in every RX_half_frame, each node in PIP hops to a new frequency or channel, given by a hopping sequence. Such a frequency hopping sequence can be conveyed as the part of the schedule dissemination from the sink to the source. The schedule dissemination also ensures that a transmitter node knows the hopping sequence of a next hop receiver.

We now evaluate the efficacy of a simple frequency hopping scheme where a node follows a frequency sequence of {k, k+1, ..., n, 1, ...} in circular fashion. For evaluation, we setup a linear topology of 10-nodes using channel 12 (2.410 GHz) to channel 21 (2.455 Ghz) of 802.15.4 channel spectrum. To introduce interference, we use a WiFi source operating on channel 6 (most prevalent channel used in deployments operating at a center frequency of 2.426 Ghz). Note that this 20 MHz WiFi channel causes interference on 5 different 5 MHz 802.15.4 channels. The WiFi source generates 1777 byte packets at 6 Mbps (channel occupancy of $\simeq 2.3ms$) every 'x' ms. Apart from this, to introduce realistic WiFi interference, we use WiFi traces from [31]. We extract the packet size and inter-packet time from these traces to generate the required WiFi interference. The PIP source transmits 1000 packets of data and the end-to-end transfer lasts for around 15 seconds.

Fig. 14 shows the performance of PIP in terms of throughput, with and without frequency hopping, for varying intensity of interference. For inter-packet time of $4ms$ which is a "very intense level" of interference, throughput without frequency hopping dips to 1.3 Kbps from 60 Kbps whereas frequency hopping achieves a throughput of 16.9 Kbps. Even for the extreme case where we send back-to-back WiFi packets (with only the minimal DIFS, configured to $40\mu sec$), we get a throughput of 14.5 Kbps with frequency hopping, while the throughput without frequency hopping drops to 0.

This shows that frequency hopping is indeed effective in mitigating the effect of interference. When interference is introduced using real WiFi traces, PIP with frequency hopping shows x1.6 throughput improvement compared to PIP without frequency hopping. Also, the throughput is reduced by only $\simeq$ 10 Kbps to $\simeq$ 50 Kbps from 60 Kbps as compared to by $\simeq$ 30 Kbps without frequency hopping. Thus PIP with a simple frequency hopping mechanism works quite effectively even in intense interfering environments.

4.10 Comparing PIP, Flush, Fetch for the Volcano Monitoring application

Sec. 4.6 & 4.7 examined PIP's performance and compared it with the *ideal*, or *maximum* possible. We showed that PIP as a protocol achieves close to ideal utilization. We now examine how PIP compares with Flush [6] and Fetch [5]: two prominent transport protocols intended for bulk transfer applications.

For this, we undertake a *thought experiment* based on various experimental numbers: how would it have helped had PIP been used for the Volcano Monitoring application? While only a real deployment can answer the question in a true sense, the numbers we present are meant to be instructive. We consider the 16-node topology deployed to detect seismic activities for Volcano Monitoring as described in [5]. This topology has 1, 6, 5, 2, 1, and 1 node(s) at distance 1, 2, 3, 4, 5, 6 hop(s) from the sink node respectively. [5] reports that 52736 bytes of data are downloaded from each node, one at a time, after each seismic event.

The first column under PIP, in Tab. 4, shows the through-

[6] Since we collect the log of queue length in memory, we run out of the available 10 KB memory beyond a certain log size.

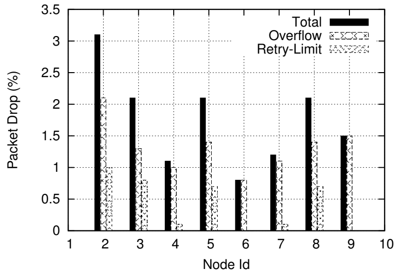

Figure 12. Packet Drop Statistics at Nodes

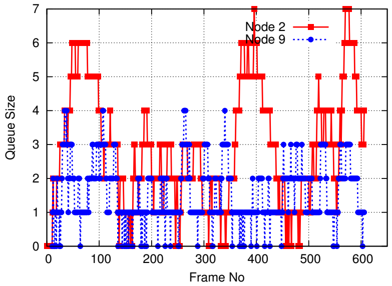

Figure 13. Queue Size variation at Nodes 2 and Nodes 9

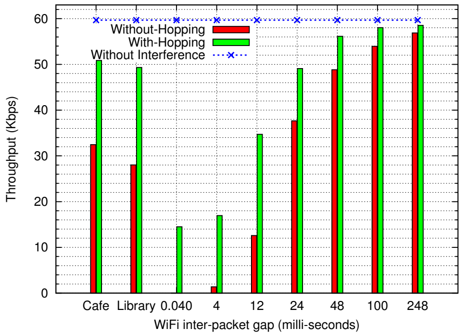

Figure 14. Performance of frequency hopping

put that would have been achieved at each hop, had PIP been used. For this, we have used numbers from Fig. 11, for the 20% error case. That is, assuming that each link has a 20% wireless error rate ([5] does not report the per link error statistics). The next column computes the latency for data transfer for each node, as $\frac{52736}{throughput}$. The last row summarizes the total latency for downloading all the data per seismic event.

For comparison, the table also shows similar numbers for Flush and Fetch. For Flush, we use the transfer-phase throughput numbers approximated from Fig-12 in [6]. And for Fetch, we directly use the latency numbers reported in Sec 6.2 of [5] (94 sec for 1-hop, 409 sec for 6-hops, 63 sec per additional hop). Now, reducing data transfer time is important for the Volcano Monitoring application, since as [5] observes, during the period of transfer, nodes had to stop sampling so as to ensure that the data being transferred was not overwritten. Thus the latency of the data transfer affected the node's ability to record back-to-back events.

The last line of Tab. 4 shows the total time that would require to collect the data from all the 16 sensor nodes using a particular transport protocol. For e.g for PIP it is $(9.0*1)+(9.2*6)+(9.4*5)+(9.5*2)+(9.6*1)+(9.8*1) = 149.6$ seconds. We see that Fetch requires almost 1 hour to collect data which, as [5] observes, is due to the delay required to ping each node and the intra-path interference during data transfer. Flush requires much less time, 15 minutes, due to it's rate-control mechanism to deal with intra-path interference. PIP requires the least time, 150 seconds, to collect the 52736 bytes of data from all nodes. This is 6 times less than Flush and 23 times less than Fetch[7] This shows the effectiveness of the multi-channel TDMA operation over multiple-hops. The individual time required for a node to transfer the data is under 10 seconds irrespective of it's hop, which implies that PIP minimally affects the node's ability to sense the volcanic events, in this particular application.

Before we explain the results in the table, we point out few important observation as mentioned in [5]. In [5], during the period of transfer, nodes had to stop sampling so as to ensure that the data being transferred was not overwritten. Thus the latency of the data transfer affected the node's ability to record back-to-back events. Further, this latency varied with the depth of the node in the routing tree. This, authors observed, was due to the increased delay for propagating Fetch command messages and the increasing packet loss and retransmission overheads as the data flows over multiple hops to the base. In this regard, we now compare PIP with Flush and Fetch.

4.11 PIP versus Flush: simulation based comparison

We now present a simulation-based comparison of PIP with Flush [6]. For this, we have extended our simulator to simulate Flush in addition to PIP. Fig. 15 compares the transfer-phase throughput achieved using PIP versus that achieved using Flush. The various plots in the graph show the cases of 0%, 10%, and 20% error rate for all links. We have used a 112-byte payload for Flush, counting out only the 12-byte 802.15.4 header and the 4-byte timestamp from the 128-byte maximum possible with CC2420. Whereas we have assumed a 103-byte payload for PIP, as per the calculations in Sec. 4.3.

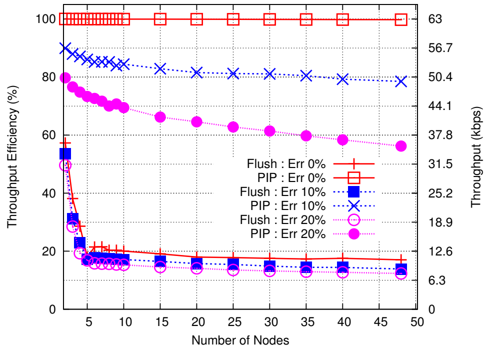

Figure 15. PIP vs Flush: simulation comparison

The right-side and left-side y-axes show the throughput and the efficiency, as explained earlier. The x-axis is shown until a maximum of 48 nodes, as in Fig-21 in [6]. We observe that under most conditions, PIP outperforms Flush by a huge margin. For the 10% error rate case, for a 48-node path, PIP gives a throughput of 49.3 Kbps while Flush gives 8.7 Kbps. This represents a factor of 5.6 improvement over Flush. The similar numbers for the 20% error rate case represent a factor

[7] Note that in the above comparison, we have assumed 20% emulated error rate for PIP, but no emulated error rate for Flush. If we assume healthy links, i.e. 0% emulated error rate, then PIP takes only 107 sec to complete the transfer: 8 times and 32 times better than Fetch and Flush respectively.

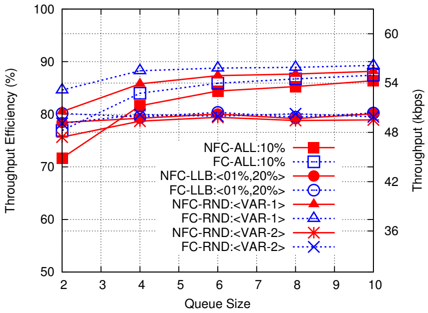

Figure 16. PIP performance with and without flow control

of 4.5 improvement over Flush.

In our simulation, Flush's throughput over a 48-node path is 10.7 Kbps; this is higher than that reported in Fig-21 of [6] primarily due to our use of a larger payload in each packet. To confirm this, we ran the simulation for Flush, using a 35-byte payload, as in Fig-21 of [6], and obtained a throughput of $\simeq$ 700 bytes/sec, which roughly matches the $\simeq$ 600 bytes/sec reported in Fig-21 of [6]. This also serves as a consistency check that our Flush simulator is indeed behaving right. As a further consistency check, note that the shape of the throughput-versus-hops plot using our simulator roughly matches Fig-21 of [6].

4.12 Non-requirement of flow-control

It is tempting to consider having flow-control in the design of PIP to accommodate various wireless errors, much like Flush [6] has considered. However, providing such a back channel is not simple in the hardware platform we use while maintaining pipelining. A different platform which permits piggybacking flow control information on the link-level ACKs could easily achieve this. However, as we show in this section, not having flow control does not affect PIP performance much.

To show that PIP performs well despite no flow control, we take the help of the simulator. Flow control is implemented in the simulator in a rather optimistic fashion; a node "magically" knows whenever its downstream node's queue is full, and does not transmit during that frame Fig. 16 shows the overall throughput (including exchange of SNACK/EOF) as a function of the queue size for various error rates. In this figure, NFC stands for no-flow-control and FC stands for flow-control. We present four scenarios. In the first case (ALL), all the links have the same error rate of 10%. In the second case (LLB), the last hop of the path is the bottleneck with 20% error rate, other links have uniform 1% loss rates. In the third and fourth case (RND), we randomly choose for each link an error rate from the set {0,1,5,10,20}.

We see in the graphs that in all cases, there is very little benefit to flow control, especially with a queue size of 10. The reason for this is that in PIP, having no flow control results in queue losses. But with flow control, there is an equivalent slow-down at the source. The dropped packets due to queue overflow no doubt would have to be retransmitted all the way from source to the sink. For most considered error profiles, this results in an extra round or two of EOF-SNACK exchange for the NFC case. Given the pipelined slotted mode of operation, this overhead turns out to be quite small in PIP.

We have tested various other similar scenarios too, with other error profiles, with the same overall result. We have also verified the same overall result using the Markov analysis (not presented here for lack of space).

5 Discussion

One active flow vs multiple active flows: In the solution approach to the problem involving bulk transfer of data, one could consider multiple simultaneous flows. Such an approach would mean a considerably complex MAC (much like IEEE 802.16) including mechanisms for (a) gathering information from all nodes to construct interference maps (b) scheduling algorithm to compute link transmission schedules (c) dissemination of the computed schedule to all nodes, apart from (d) global time synchronization.

Such an approach would be worth the effort if the performance gains are substantial. What is indeed the performance gain of having multiple flows, over the extended PIP case (Sec. 3.6) where two flows are active? When two flows are active, the sink is always busy. So even if more flows are supported, at best, these additional flows may be able to bring their data one-hop away from the sink. So the performance gain experienced by these additional flows, would at best be the performance gain of a single-hop data transfer as opposed to a multi-hop transfer. Now, Figure. 11 shows that for the no error case, the throughput improvement over one hop as opposed to 9 hops is only 1%. For the 20% error case, the improvement is about 10% (i.e gain of 6Kbps over the base 44Kbps). We feel that this performance improvement is too small, to warrant a complex MAC supporting an arbitrary number of simultaneous flows.

Time synchronization requirement: The literature on multi-hop time-synchronization is quite elaborate. TDMA-based MAC approaches have in the past been rejected since synchronization is difficult to achieve. While in the generic sense it may be hard to achieve, we have still been able to effectively use a TDMA-based system for two reasons. First, we require synchronization only during data transfer and not always. Second, only the synchronization error between neighbouring nodes matters. If we can achieve a bound on this, that is good enough to set a small guard time. In fact, in our implementation, the guard time is dominated by the channel switching time rather than the synchronization error.

Generic MAC design: We have designed the PIP MAC to provide a connection-oriented primitive explicitly for high throughput unidirectional data transfer, which applies for a class of sensor network applications. After building the prototype and our evaluation on it, we believe that a generic MAC can be designed along the same lines, to support generic data flows. The reusable components of our working system would potentially be: the centralized connection-oriented architecture, multi-hop synchronization piggybacked on periodic packets, and multi-channel operation. While our focus has been on sensor network applications involving 802.15.4 nodes, the MAC design is applicable with other technology choices available in the 900MHz band or even with 802.11.

6 Conclusion

In this paper, we considered the goal of achieving high throughput bulk data transfer. To this end, we have presented a novel multi-hop connection oriented MAC primitive (PIP) and built on top of this primitive a highly efficient centralized multi-channel TDMA system. In general, multi-hop time-synchronization is considered hard to achieve. But PIP uses synchronized time-slots effectively by (a) requiring synchronization only during data transfer, (b) embedding synchronization information in each data packet, and (c) by designing to be concerned only about synchronization error between neighbours.

We demonstrated the feasibility of our ideas via an actual implementation and effectiveness of our solution through extensive evaluation on a network with up to 9 hops. Our results show over twelve times better throughput than the current state-of-the-art approach. Apart from good performance, PIP also has several nice properties, such as being simple, easy to analyze or implement. Importantly, it does not require flow control to accommodate variable wireless error rates.

We believe that our contribution also lies in showing the practicality of a multi-hop TDMA wireless system, which we hope will have implications beyond the specific application environment considered in this paper.

Acknowledgment

This work was supported in part by the India-UK Advanced Technology Centre (IU-ATC) project. The simulations for Flush in Sec. 4.11 were carried out by Piyush Joshi. We thank the reviewers as well as the shepherd, Prof. Adam Wolisz, for their critical comments which helped improve our work substantially.

7 Additional Authors

8 References

[1] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon, “Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks,” in *IPSN*, Apr 2007.

[2] J. Paek, K. Chintalapudi, J. Cafferey, R. Govindan, and S. Masri, “A wireless sensor network for structural health monitoring: Performance and experience,” in *EmNets-II*, may 2005.

[3] K. Chebrolu, B. Raman, N. Mishra, P. K. Valiveti, and R. Kumar, “BriMon: A Sensor Network System for Railway Bridge Monitoring,” in *MobiSys*, Jun 2008.

[4] G. Werner-Allen, K. Lorincz, M. Welsh, O. Marcillo, J. Johnson, M. Ruiz, and J. Lees, “Deploying a Wireless Sensor Network on an Active Volcano,” *IEEE Internet Computing*, Mar/Apr 2006.

[5] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, “Fidelity and Yield in a Volcano Monitoring Sensor Network,” in *OSDI’06*, Nov 2006.

[6] S. Kim, R. Fonseca, P. Dutta, A. Tavakoli, D. Culler, P. Levis, S. Shenker, and I. Stoica, “Flush: A Reliable Bulk Transport Protocol for Multihop Wireless Networks,” in *Sensys*, Nov 2007.

[7] F. Osterlind and A. Dunkels, “Approaching the Maximum 802.15.4 Multi-hop Throughput,” in *HotEmNets*, Jun 2008.

[8] F. Stann and J. Heidemann, “RMST: Reliable Data Transport in Sensor Networks,” in *Proceedings of the First International Workshop on Sensor Net Protocols and Applications*, Apr 2003.

[9] Y. Sankarasubramaniam, O. Akan, and I. Akyildiz, “ESRT: Event-to-sink reliable transport in wireless sensor networks,” in *Mobihoc*, Jun 2003.

[10] C.-Y. Wan, A. T. Campbell, and L. Krishnamurthy, “PSFQ: a reliable transport protocol for wireless sensor networks,” in *WSNA*, 2002.

[11] “IEEE 802.15 WPAN Task Group 4 (TG4),” http://www.ieee802.org/15/pub/TG4.html.

[12] “IEEE P802.11, The Working Group for Wireless LANs,” http://grouper.ieee.org/groups/802/11/.

[13] J. Polastre, J. Hill, and D. Culler, “Versatile Low Power Media Access for Wireless Sensor Networks,” in *SenSys*, Nov 2004.

[14] M. Li, D. Agrawal, D. Ganesan, and A. Venkataramani, “Block-switched Networks: A New Paradigm for Wireless Transport,” in *NSDI’09*, Apr 2009.

[15] V. Rajendran, K. Obraczka, and J. GarciaLunaAceves, “EnergyEfficient, CollisionFree Medium Access Control for Wireless Sensor Networks,” in *SenSys*, Nov 2003.

[16] L. van Hoesel and P. Havinga, “A Lightweight Medium Access Protocol (LMAC) for Wireless Sensor Networks,” in *INSS*, 2004.

[17] A. Rao and I. Stoica, “An Overlay MAC Layer for 802.11 Networks,” in *Mobisys*, Apr 2005.

[18] A. Rowe, R. Mangharam, and R. Rajkumar, “RT-Link: A Time-Synchronized Link Protocol for Energy-Constrained Multi-hop Wireless Networks,” in *SECON*, 2006.

[19] J. Mo, H.-S. W. So, and J. Walrand, “Comparison of Multichannel MAC Protocols,” *IEEE Transactions on Mobile Computing*, May 2007.

[20] K. S. J. Pister and L. Doherty, “TSMP: Time Synchronized Mesh Protocol,” in *Distributed Sensor Networks (DSN 2008)*, Nov 2008.

[21] “IEEE 802.16 WirelessMAN,” http://www.ieee802.org/16/.

[22] B. Hohlt, L. Doherty, and E. Brewer, “Flexible Power Scheduling for Sensor Networks,” in *IPSN’04*, Apr 2004.

[23] N. Ramanathan, M. Yarvis, J. Chhabra, N. Kushalnagar, L. Krishnamurthy, and D. Estrin, “A Stream-Oriented Power Management Protocol for Low Duty Cycle Sensor Network Applications,” in *EmNetS-II*, 2005.

[24] N. W. McKeown, “Scheduling Algorithms for Input-Queued Cell Switches,” Ph.D. dissertation, University of California at Berkeley, 1995.

[25] D. Koutsonikolas, T. Salonidis, H. Lundgren, P. LeGuyadec, Y. C. Hu, and I. Sheriff, “TDM MAC Protocol Design and Implementation for Wireless Mesh Networks,” in *CoNEXT*, Dec 2008.

[26] P. Djukic and P. Mohapatra, “Soft-TDMAC: A Software TDMA-based MAC over Commodity 802.11 hardware,” in *INFOCOM’09*, Apr 2009.

[27] B. Raman and K. Chebrolu, “Censor Networks: A Critique of “Sensor Networks” from a Systems Perspective,” *CCR*, Jul 2008.

[28] D. Cox, E. Jovanov, and A. Milenkovic, “Time Synchronization for ZigBee Networks,” in *SSST*, Mar 2005.

[29] H.-S. W. So, G. Nguyen, and J. Walrand, “Practical Synchronization Techniques for Multi-Channel MAC,” in *Mobicom*, Sep 2006.

[30] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi, “The Flooding Time Synchronization Protocol,” in *SenSys*, Nov 2004.

[31] “Traces of Stanford CS department’s wireless network.” http://crawdad.cs.dartmouth.edu/stanford/gates.