Compressing MAC Headers on Shared Wireless Media

Jesus Arango[1], Matthew Faulkner[2], and Stephen Pink[2]

[1] Cisco Systems, USA
jearango@cicso.com
[2] Computing Department, Lancaster University, Lancaster, UK
{faulknem,pink}@comp.lancs.ac.uk

Abstract. This paper presents a header compression algorithm that unlike previous protocols is capable of compressing MAC headers in a multiple-access (shared) channel. Previous schemes could not compress MAC headers because the address fields are needed to identify the intended destination as well as the intended context space under which compressed headers are to be interpreted.

We approach this problem by sharing a single context space among participating nodes. The compression context for a new flow between two nodes is initialized and synchronized by transmitting an uncompressed frame with a context label that is randomly selected by the sender. Frames with compressed MAC headers will have no address fields. A receiving node that has a context which matches the label in a compressed frame will be able to decompress the header by expanding the label into the corresponding fields stored in the context. Mechanisms are presented to address label conflicts where a node is receiving compressed frames with the same label from multiple senders.

We evaluate our work by simulating an 802.11 network that implements the header compression algorithm. The simulation results show that when MAC headers are compressed there is throughput improvement of up to 15% in the experiments we conducted. This is in addition to the throughput improvement achieved by compressing IP, TCP, UDP and higher level headers.

1 Introduction

Traffic studies [10,2] have shown that approximately 50% of IP packets are 80 bytes long or less. The overwhelming majority of these packets have at least 40 bytes of headers, resulting in poor efficiency as a consequence of small payloads and large header sizes.

Low efficiency is a major concern on low bandwidth channels, and in particular wireless networks where bandwidth is scarcely enough to meet application demands. The maximum data rate for off-the-shelf technologies such as 802.11 has increased considerably. However, network deployments requiring a larger transmission range or lower energy consumption must unavoidably fall back to less complex modulation schemes that result in lower data rates.

J. Zheng et al. (Eds.): ADHOCNETS 2009, LNICST 28, pp. 576–591, 2010.

Multiple header compression protocols have been proposed to compress network and transport-level headers [8,5,3,1]. Unlike previous protocols, the algorithm presented in this paper is capable of compressing MAC headers in a multiple-access (shared) channel.

Line efficiency is the fraction of transmitted data that is not considered overhead; in other words, the fraction of transmitted data that has some explicit, useful purpose to the application. Line efficiency is formally defined as:

$$efficiency = \frac{payload}{headers + payload} \tag{1}$$

Header compression improves channel efficiency by exploiting redundancies between header fields within the same packet or consecutive packets belonging to the same packet stream. It works by sending static fields only initially and utilizing dependencies and predictability for other fields. Compression performance depends in part on the optimal classification of the aggregate packet stream at the compressing entity into sub-streams of highly correlated packets. Correlation within a packet stream implies a highly predictable pattern of changes between consecutive packets, or that large sections of the header chain remain constant across packets of the same stream, or both. In practice, the aggregate packet stream is usually classified into network-level or transport-level flows, as packets belonging to the same flow are highly correlated. For example, many fields such as addresses, ports and protocol types remain constant across all packets of the same flow, and changes in other fields are highly predictable.

Accordingly, each flow is compressed independently by first sending a packet with full, uncompressed headers to establish a *context* that provides common knowledge between sender and receiver about static field values as well as initial values for dynamic fields. This stage is known as *context initialization*. Subsequent compressed headers are interpreted and decompressed according to a previously established context. Constant fields need not be sent with compressed headers. Fields that change predictably, such as sequence numbers, are encoded incrementally, requiring only a small number of bits. Fields with random, unpredictable changes, for example checksums, must be sent in every header.

A *context label* must be transmitted explicitly or implicitly with every header. For context initialization headers, the label determines the context being initialized. For compressed headers, it determines the context used to interpret the compressed data. The context label should be short enough to provide efficient compression but long enough to support an adequate number of flows.

In some situations, the number of active flows will inevitably exceed the number of contexts. A context replacement strategy such as *Least Recently Used* (LRU) is necessary in such situations. Context thrashing occurs when headers are seldom matched with an existing context and have to be sent uncompressed. Techniques such as hysteresis ensure that packet streams with the highest compression rates retain their context. Such techniques are more likely to be needed in the middle of the network.

Encoding dynamic fields in fewer bits using incremental encoding techniques requires periodic and synchronized context updates. Not all packets need to trigger a context update, but the more seldom the context is updated the more bits that are usually needed for dynamic fields. Packets that trigger a context update are referred to as *context-updating packets*. Failure to update receiver context as a result of packet loss will lead to context inconsistencies and incorrect decompression of subsequent packets. This is known as *loss propagation*. Incorrect updates of receiver context as a result of *residual errors*[1] will similarly lead to incorrect decompression of subsequent packets. This is known as *error propagation*.

The number of packets discarded due to incorrect decompression as a result of loss of synchronization depends on the channel's error rate and round-trip time. The higher the error rate the greater the probability of losing a context-updating packet. The round-trip time determines the number of packets that are discarded before feedback can be sent to re-synchronize the context.

Early work on header compression [8,5] focused on slow serial links and wired networks with low error rates. Their encoding and feedback mechanisms are not robust enough to reduce, prevent and correct loss of synchronization. Therefore, they do not perform well on channels with high error rates and long round-trip times. More recent [1] work has addressed loss and error propagation issues associated with high error rates and long round-trip times commonly found in wireless links. The algorithm presented in this paper , unlike previous protocols, is capable of compressing MAC headers on shared wireless media.

In classical header compression schemes, a context space is kept synchronized between a *compressor* (sender) and a *decompressor* (receiver). Each context space is thus associated with a *logical channel* represented by an ordered *(sender, receiver)* pair. Header compression has traditionally been performed on point-to-point links. By definition, point to point links consist of a single logical channel in each direction. Interpreting context labels transmitted on point-to-point links is a straightforward process because any transmitted label is unambiguously associated with only one logical channel.

Traditional header compression protocols can be extended to support shared media by maintaining multiple context spaces and relying on link-level addressing to correctly associate each packet with the correct logical channel. The destination address ensures delivery to the intended node. The source address ensures decompression with the correct *context table*. Accordingly, MAC headers may not be compressed when link-layer addressing is used to represent logical channels.

The alternative is to have a single logical channel, and the critical issue is to consistently share a single context set among multiple network nodes. Under strict consistency, each node wishing to initialize a header compression context must select a label that is not currently in use by any other node[2], and all nodes must agree on this selection.

Strict consistency is equivalent to the *consensus problem* [4], where a set of processes (nodes) must agree on a value after one or more of the processes have proposed what the value should be. Achieving consensus is an expensive operation

[1] Residual errors are errors that go undetected by error detecting codes.

[2] Under a finite label space, such selection must involve an arbitration mechanism if no free labels are available.

and, efficiency-wise, is considered a difficult problem. Known algorithms require several rounds of processes exchanging values among themselves by using group communication. Such solutions also rely on group communication primitives that are more powerful than the broadcast services commonly available in LAN technologies. An important result by Fischer *et al.* [6] also demonstrated that no algorithm can guarantee to reach consensus in an asynchronous system[3] where process failures are considered.

With the above considerations in mind, it is imperative to determine the least degree of consistency required such that any two adjacent nodes can compress message headers through the use of synchronized contexts. A strict context synchronization with global scope seems unnecessary since synchronized context only need to be kept between adjacent nodes. We argue that any consistency scheme is sufficient if the following invariant holds:

If any two nodes use the same context label to decompress messages, then they should not receive messages bearing this label whose intended destination is the other node.

Let R be the *compression relation* where $(x, y, l) \in R$ if x compresses messages for y using label l. Let $G = (V, E)$ be the graph representing the network connectivity, where V is the set of nodes and E is the set of edges. As always, two nodes are connected by an edge if they are within range. The previous invariant can be formally expressed as follows.

$$\forall m, n, p, q, l[R(m, n, l) \wedge R(p, q, l) \rightarrow (m, q) \notin G \wedge (p, n) \notin G]$$

The algorithm presented in this paper shares a single context space among participating nodes. The compression context for a new flow between two nodes is initialized and synchronized by transmitting an uncompressed frame with a context label that is randomly selected by the sender. Frames with compressed MAC headers will have no address fields. A receiving node that has a context which matches the label in a compressed frame will be able to decompress the header by expanding the label into the corresponding fields stored in the context. Mechanisms are presented to address label conflicts where a node is receiving compressed frames with the same label from multiple senders.

We evaluate our work by simulating an 802.11 network that implements the header compression algorithm. The simulation results show that when MAC headers are compressed there is throughput improvement of up to 15% in the experiments we conducted. This is in addition to the throughput improvement achieved by compressing IP, TCP, UDP and higher level headers.

The rest of the paper is organized as follows. Section 2 presents the algorithm for compressing MAC headers on shared wireless media. Section 3 discusses the implementation of the algorithm on 802.11 networks. Section 4 presents the experimental results and Section 5 provides some concluding remarks.

[3] An asynchronous system is that which imposes an upper bound on processing and communication delays. That is, processes are entitled to assume that another process is faulty if it does not respond within a specified timeout period.

2 MAC Header Compression

This section presents a header compression algorithm capable of compressing MAC headers on shared wireless media. The central concept behind the algorithm is that a node compressing a flow with label l should be the only transmitter using label l within range of the intended destination r. The algorithm aims to avoid and correct situations where two or more transmitters within range of r compress with label l.

The algorithm is presented by walking through the compression cycle and making references to key parts of the pseudo-code. The compression routine is shown in Algorithm 1 while Algorithm 2 illustrates the decompression routine. There is an additional routine (Algorithm 3) that, as described later, is crucial in resolving label conflicts.

2.1 Compression

The compression routine maintains a *transmission context set* (1.1) that contains the *transmission contexts* currently used for compression. At a minimum, each transmission context stores a label, a *flow key* that uniquely identifies a flow, and a *compression state* that conveys whether the context is in the initialization stage or some other stage. A transmission context may optionally contain other media-dependent information to compress predictable fields such as sequence numbers.

The compression routine also uses a set of *unavailable labels* (1.2). As described later, this set contains the labels that have been previously observed on the channel. As such, the set of unavailable labels is actually updated by the decompressor.

To compress a frame, the transmission context set T must first be searched for a corresponding context (1.4). This is usually done by forming a key based on the flow-defining fields of the frame and searching for a context with a matching key.

If a context exists (1.5), then that context is used to compress the frame. The context's state may also need to be updated and this is done differently depending on whether the MAC layer uses synchronous[4] acknowledgements (i.e. 802.11). The key observation here is that the context must remain in the initialization (INIT) state until the compressor is fairly confident that the decompressor has received the static information correctly. With synchronous acknowledgements, all that needs to be done here is to store a reference to the last context used (1.9). This reference will be used by the decompressor to update the context's state when the next acknowledgement arrives. Otherwise, the state must be updated here (1.7) based on a unidirectional mode that implements a state machine where transitions are performed only on account of periodic timeouts or frame counters associated with each state.

[4] Synchronous in this context means that the acknowledgement is expected to be transmitted at a precise time slot.

Algorithm 1. COMPRESSION

```
 1: $T$: local context table
 2: $U$: set of unavailable labels
 3: procedure COMPRESS(frame $f$)
 4:     if $T$ has a context for $f$'s flow then
 5:         compress $f$ using existing context
 6:         if unidirectional mode then
 7:             update the state of the context
 8:         else
 9:             $lastCxt \leftarrow$ context reference
10:         end if
11:     else
12:         Select a random label that is not in $T$ nor $U$
13:         create new context $l$ with state set to INIT
14:         insert context $l$ into $T$
15:         compress $f$'s flow using label $l$
16:     end if
17: end procedure
```

If a context is not found, a new context is created with a label that is randomly generated such that it is not contained in T or the set U of unavailable labels (1.12). The context is inserted into T and its state is set to INIT. Finally a context initialization frame is sent after appending the label to the original frame.

2.2 Decompression

The most interesting processing occurs in the decompressor. The precise process depends on whether the received frame is a context initialization frame, a compressed frame, or as described later, a conflict notification frame.

Context Initialization Frames. The first thing to do with a context initialization frame is to check for errors (2.6) and discard the frame if the redundancy check fails (2.7). This is accomplished by transmitting some sort of redundancy code that is application dependent. Our algorithm is generic in nature and does not advocate a specific redundancy code. However, many MAC protocols use *Cyclical Redundancy Check (CRC)* and the term CRC is used in this paper to denote redundancy codes in general.

The next step is to compaper the frame's destination address with the local node's address (2.8). A mismatch indicates the local node is not the intended recipient. However, the frame's label should be made locally unavailable by adding it to set U (2.9), thus preventing the local compressor from creating a conflict at the intended recipient of this flow. Note that the reception of this message only confirms that we are within range of the sender. However, the likelihood of being within range of the intended receiver given that we are within range of the sender is quite high.

Further processing at this point (2.11) is only performed for locally addressed, error-free frames. The decompression context set R must now be searched for a matching context (2.12). If no context is found a new one is created (2.19).

If a context is found, it must also be determined if the source address in the frame matches the source address in the context (2.13). This comparison is important because the existence of a local context alone does not necessarily indicate a conflict. A match in the source addresses is an indication that *acknowledged transmitter* has sent another context initialization frame. The acknowledged transmitter refers to the node who sent the first context initialization frame that triggered the creation of the local context. Multiple context initialization frames may be sent because the acknowledged transmitter either wants to change the context or because it is operating in unidirectional mode and thereby periodically refreshing the context. The decompressor must therefore update the contents of the context (2.16).

If the source addresses do not match we clearly have a conflict in the making and must therefore send (2.14) a *collision notification frame* $CNF(l, addr)$, where l is the conflicting label and $addr$ is the address of the acknowledged transmitter stored in the local context. A $CNF(l, addr)$ alerts all nodes in the local neighborhood except the acknowledged transmitter $addr$ to cease and desist from transmitting label l. CNF messages are sent asynchronously like any other data message. A synchronous acknowledgement would likely cause a collision at the offending transmitter because the offending frame is likely to be acknowledged by a third party.

Compressed Frames. The first thing to note about compressed frames is that we cannot immediately perform a CRC verification, as the CRC is computed prior to compression. Consequently, the receiver must perform the CRC validation after decompression. Moreover, special processing of CRC failures may be required if CRC checks are used to detect label conflicts.

The first order of business is to determine if R contains a matching context for the frame's label (2.24). If no context is found we have no business with this frame other than adding its label to set U to prevent the local compressor from using this label. Finding a context, on the other hand, allows the decompression of the frame (2.25) by expanding its label with the information stored in the context. A CRC verification is conducted (2.26) once the frame is decompressed. A successful CRC verification indicates that both the transmitter and receiver are referring to the same context, resulting in a successful decompression of the frame.

Note however that a CRC failure can be caused by either a transmission error or a label conflict. When compressed frames are received bearing a label for which a local context exists but sent by a node other than the one indicated in the context, then decompressing the frame will expand the label with incorrect information, resulting in a CRC failure.

Several mechanisms are possible to detect label conflicts and correct this situation. For MAC protocols with synchronized acknowledgements, a conflicting sender will receive no acknowledgements and will make several transmission

Algorithm 2. DECOMPRESSION

```
$R$: local context table
$U$: set of unavailable labels
procedure DECOMPRESS(frame $f$)
    $l \leftarrow label(f)$
    if $f$ is a context initialization frame then
        if $f$ fails CRC then
            drop $f$
        else if $destination(f)$ != local address then
            insert $l$ into $U$
            drop $f$
        else
            if $R$ has a context $ctx$ for $l$ then
                if $source(ctx)$ != $source(f)$ then
                    send $CNF(l, source(ctx))$
                else
                    $key(ctx) \leftarrow key(f)$
                end if
            else
                insert new context $l$ into $R$
            end if
            decompress $f$
        end if
    else if $f$ is a compressed frame then
        if $R$ contains a context for $l$ then
            decompress using context $l$
            verify CRC
            update CRC verification history
            if CRC verification failed then
                drop $f$
                if more than $k$ out $m$ frames fail CRC then
                    send CNF
                end if
            end if
        else
            add $l$ to $U$
            drop $f$                                   ▷ not the intended receiver
        end if
    else if $f$ is a CNF then
        $ReceiveCNF(f)$
    end if
end procedure
```

attempts before giving up, removing the context and randomly selecting a new label. If no acknowledgements are used, unidirectional mode forces all local senders to periodically revert to a state where uncompressed frames are sent and conflicts can be easily detected at the receiver.

Conflict resolution is also possible when synchronous acknowledgements are not available and unidirectional mode is not desired. *Asynchronous* acknowledgements are introduced for context initialization frames only to ensure that the context is initialized at the receiver. This conflict resolution scheme is shown in Algorithm 2. It is based on the observation that an abnormal number of CRC failures is likely to be an indication of a label collision because the occurrence of so many transmission errors would be rather unlikely. Statistical analysis can be used to formally determine what exactly is an abnormal amount of CRC failures.

Each decompression context has a circular buffer that stores the history of the CRC outcome for the last k frames, with each outcome being either a *success* (S) or a *failure* (F). Assume that all failures are due to transmission errors and let X be the number of failures in the last k frames. Let $m < k$ be some bound on the number of transmission errors such that

$$\Pr[X > m] < \epsilon \tag{2}$$

where ϵ is an arbitrarily small probability bound. After updating the CRC history (2.27) a new value of X will be observed. If the CRC fails, the frame is dropped (2.29) and a label conflict is assumed if $X > m$ (2.30).

Note that ϵ is a fixed parameter. The value of m can be determined with Chernoff's bound:

$$\Pr[X > (1+\delta)\mu] < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu} \tag{3}$$

Where $\delta > 0$ and $\mu = E[X]$. By letting $m = (1+\delta)\mu$ the bound can be rewritten as:

$$\Pr[X > m] < \left[\frac{e^{\left(\frac{m}{\mu}-1\right)}}{\left(\frac{m}{\mu}\right)^{\left(\frac{m}{\mu}\right)}}\right]^{\mu} \tag{4}$$

From the following inequality

$$\left[\frac{e^{\left(\frac{m}{\mu}-1\right)}}{\left(\frac{m}{\mu}\right)^{\left(\frac{m}{\mu}\right)}}\right]^{\mu} \leq \epsilon \tag{5}$$

we can ensure by transitivity that $\Pr[X > m] < \epsilon$. The only term in (5) that has not yet been defined is μ:

$$\mu = E[X] = \sum_{i=1}^{k} 1 - (1-p)^{n_i} \tag{6}$$

where p is the bit error rate and n_i is the size of the ith frame. To determine m, one would first set k to a fixed value and minimize m subject to the constraint in (5). For example, Figure 1 plots m as a function of k for $\epsilon = 0.02$, a bit error rate of 10^{-4} and an average frame size of 500 bytes.

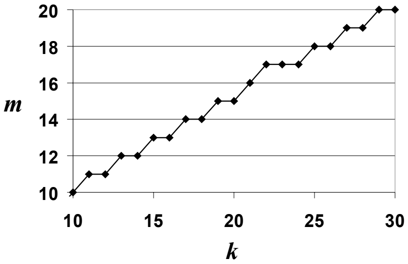

Fig. 1. Parameter m as a function of parameter k, for $\epsilon = 0.02$, $p = 10^{-4}$, and $n_i = 500$ for all i

Conflict Notification Frames. Nodes that receive a CNF should only process it if the CRC verification succeeds (3.2). Receiving nodes must check whether the address contained in the CNF matches their local address (3.3). A match indicates that the local node is the acknowledged sender for the label and should therefore ignore the CNF. A mismatch indicates that we are not the acknowledged sender and should therefore search T to determine if we have an offending context for the label. If a context is found it must be removed from T (3.5). Whatever the outcome of the search, the label should be inserted into U to avoid future conflicts.

Algorithm 3. CONFLICT NOTIFICATION

```
 1: procedure ReceiveCNF(frame $f$)
 2:     if CRC verification succeeds then
 3:         if $addr(f)$ != local address then
 4:             if a context for $label(f)$ is found in $T$ then
 5:                 remove context from $T$
 6:             end if
 7:             insert $label(f)$ into $U$
 8:         end if
 9:     end if
10:     drop $f$
11: end procedure
```

3 Compressing 802.11 Headers

This section describes how 802.11 headers can be compressed using the algorithm discussed in Section 2. The simulation experiments described in Section 4 are based on the compression of 802.11 headers as described here. Only compression of data and ACK frames will be described. RTS frames can be compressed similarly to data frames, using the algorithm of Section 2. CTS frames can be compressed similarly to ACK frames, using a stateless scheme described in this section. The RTS/CTS mechanism has been shown to be generally ineffective in improving performance even in the presence of hidden terminals [7]. Furthermore, RTS/CTS is often disabled in practice.

3.1 Data Frames

Figure 2 illustrates how data frames are compressed. The field lengths of the data frame are shown in octets and the subfield lengths in the *Frame Control* field are shown in bits. The entire header is 28 octets long including the CRC field. The RA and TA address fields store the receiver and transmitter addresses, respectively.

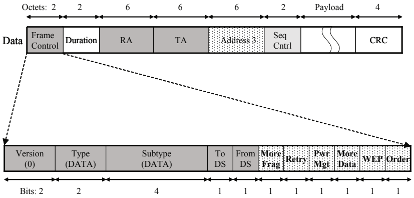

Fig. 2. Compression of 802.11 data frames

The shaded fields remain constant for a given (*receiver*, *transmitter*) pair, thus they only need to be sent initially in a context initialization frame. The dotted fields take a limited number of values. In fact, the dotted bit fields in the Frame Control rarely change at all. Rather than having a complex frame encoding where the dotted fields are sometimes transmitted and other times omitted, we decided to make them part of the flow-defining fields, along with the RA and TA address fields. The dotted fields are described below to give a better idea of how many contexts to expect between a single (*receiver*, *transmitter*) pair.

The contents of the 3rd address field depends on the type of node sending the frame as well as the mode of operation (infrastructure vs. ad-hoc). In infrastructure mode, if a station sends a frame to the access point, then the field stores the address of the destination. If the access point sends a frame to a station, then the field stores the address of the source. In ad-hoc mode, the field is fixed to the BSSID of the ad-hoc network. In practice, the number of values that this field is expected to take in infrastructure mode is just one (the default gateway)[5], as wireless clients do not usually connect to other wireless clients on the same WLAN.

All the dotted bits in the Frame Control field except the *Retry* bit will most likely always be set to a single value because they represent features that are

[5] Recall that the third address does not necessarily have to be that of a station on the same cell (Basic Service Set (BSS)). It can also be a station on another cell in the same extended service set (ESS) or any wired station on the distribution system (DS) that interconnects the cells.

rarely used or always enabled once a station associates with the network. Accordingly, there will usually be two contexts for every pair of stations, one for each value of the Retry bit.

The *Sequence Control* field can be partially compressed using delta or LSB encoding. The 802.11 MAC implements a *Stop-and-Wait, Automatic Repeat Request (ARQ)* protocol where a transmitted frame must be immediately acknowledged before proceeding to transmit the next frame. It also uses frame-level sequencing, as opposed to byte-level sequencing like TCP. These two characteristics allow the Sequence Control field to be compressed to as little as two bits. In fact, 802.11 provides an ideal environment for delta/LSB encoding as loss propagation can be fully eliminated due to the acknowledgement mechanism already in place.

3.2 ACK Frames

The underlying concept behind maintaining state in the form of compression contexts is that an unending stream of frames can be mapped into a relatively small number of header values. Only the context's label is needed to fill the missing fields in a compressed frame. Without the label the values are totally unpredictable.

For certain types of frames, however, the value of some fields is totally predictable even without maintaining state. Such is the case of ACK frames. Figure 3 shows the format of an ACK header. The designers were right not to include the address of the ACK's sender because for the receiver there is only one possible value: the destination address of the previously transmitted data frame. The *Duration* value can also be omitted in most circumstances as it is set to zero if the More Fragment bit was set to 0 in the immediately previous data frame. The Frame Control field can also be entirely omitted, with the exception of the *Power Management* bit.

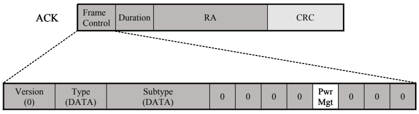

Fig. 3. Compression of 802.11 ACK frames

With some care, the RA address can also be omitted altogether. The sender includes the RA field in the computation of the CRC but omits the field in the transmitted frame. A station receiving an ACK frame will append its own address before performing the CRC check. A CRC failure indicates that either the ACK frame is corrupted or destined to someone else. There is no need to differentiate between both as the node's action is the same in both cases.

4 Experimental Results

The purpose of the experiments presented in this section is to evaluate the improvement in performance and the occurrence of label conflicts when MAC headers are compressed with the algorithm presented in this paper. The algorithm has been implemented in *ns-2* [9], a discrete event simulator with extensive support for wireless networks.

The simulated network scenario consists of an 802.11 ad-hoc network comprised of 100 nodes with a transmission range of 250 m, moving in a network space that measures 2500x750 m^2. Nodes move according to the *random waypoint* model. The *Ad-Hoc on Demand Distance Vector (AODV)*[**?**] protocol is used for multi-hop routing. The ad-hoc model was chosen over a WLAN (infrastructure) model because free interaction between any two moving nodes is likely to put more stress on the algorithm and test its correctness. That being said, the mode of operation is not a concern as the algorithm was implemented to efficiently and seamlessly support both modes.

4.1 Label Conflicts

The first experiment is designed to evaluate the impact that mobility and label size have on the frequency of label collisions. The 100 nodes in the network are divided into a group of 50 senders and 50 receivers. A one-to-one mapping is established where each sender is connected to a randomly selected receiver. This scenario was chosen to guarantee that each node had at least one active role as compressor or decompressor. However, every node is expected to be active in multiple flows because we are dealing with a multi-hop network where each node has on average 9.4 neighbors.

Each sender transmits four beacon messages per second to its corresponding receiver for a duration of 5 minutes. Compression of MAC headers is performed on each link of the path between the sender and receiver. The simulator keeps track of number of conflicts and number of contexts created during the simulation. Both are global counters that store network-wide aggregates. The number of contexts is important to determine what percentage of the contexts result in label conflicts. Note that only unique conflicts are counted as opposed to counting the number of conflicting frames. This distinction is important because the number of unique context conflicts is independent of the frame rate.

Different label lengths and node speeds were simulated to observe the number of conflicts. Ten different runs were executed for each combination of label size and speed and the results were averaged. Figure 4 shows the number of conflicts as a function of network speed for several label lengths. Some of the data points are labeled with a percentage value indicating that the number of conflicts corresponds to that percentage of the total number of contexts.

The figure shows that label lengths as short as 14 bits result in a very small percentage of conflicting flows, even for speeds that start being excessive for a network range of just 250 m. A node traveling at 35 mph will encounter a

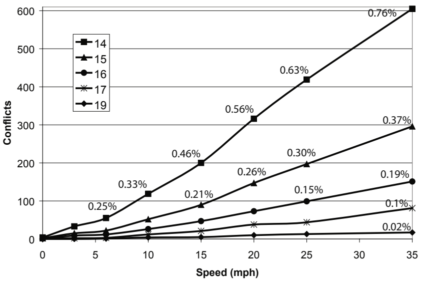

Fig. 4. Number of conflicts vs node speed for several label sizes

new neighbor on average every 2.3 seconds! Label lengths between 16 and 19 do exceptionally well, with label length 16 showing longer-trend scalability.

4.2 Performance

Header compression increases channel efficiency by reclaiming bandwidth previously consumed by headers and using it for data that is useful to the application. If the channel's capacity is sufficiently large to handle the load of uncompressed packets then there will obviously be no benefit in compressing headers.

Accordingly, an experiment was designed where flows are added one by one to increasingly saturate the channel and allow the comparison of the overall network throughput with and without header compression. Source and destination nodes are randomly selected for each additional flow. No node is chosen for more than one flow unless the number of flows exceeds half the number of nodes. Each source transmits 20 packets per second for a period of 120 seconds. The simulator returns the overall network throughput by reporting the number of packets successfully delivered to the destination nodes.

Figure 5 shows the throughput as a function of the number of flows. The throughput is shown for both compressed (enabled) and uncompressed (disabled) headers. These results correspond to a channel rate of 2 Mbps, nodes moving at 6

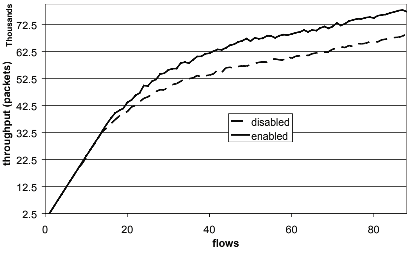

Fig. 5. Throughput vs. flows for a channel rate of 2 Mbps, nodes moving at 6 *mph* and a packet size of 60 bytes

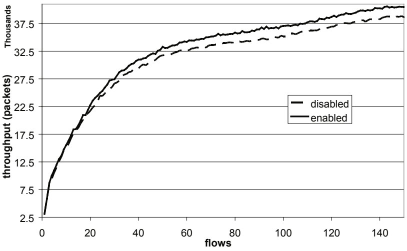

Fig. 6. Throughput vs. flows for a channel rate of 2 Mbps, nodes moving at 6 *mph* and randomly distributed packet length

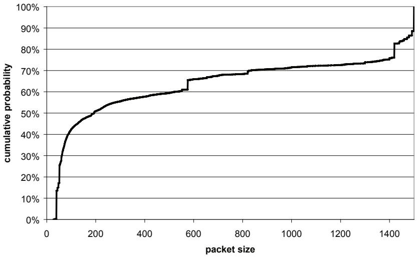

Fig. 7. Packet length distribution

mph and an IP packet size of 60, which is typical for VoIP and sensor data. The figure shows that compressing MAC headers improves throughput up to 15%.

A more generic scenario is shown in Figure 6 where nodes are moving at a speed of 6 *mph* and the IP packet size is randomly distributed according to a packet size distribution provided by Sprint Labs. The distribution is based on the analysis of the traces collected by the IPMON [10] systems on more than 30 bidirectional OC3/12/48 links. The Cumulative distribution function is shown in Figure 7. The $ns-2$ traffic generator was modified to generate packets according to this distribution.

5 Conclusions

The header compression algorithm that has been presented in this paper is capable of compressing MAC headers on shared wireless media. This is achieved by sharing a single context space between participating nodes. Context labels are randomly chosen and label conflicts are efficiently resolved with different mechanisms depending on the capabilities of the MAC protocol. The simulation results show that when MAC headers are compressed there is throughput improvement of up to 15% in the experiments we conducted. This is in addition to the throughput improvement achieved by compressing IP, TCP, UDP and higher level headers.

Acknowledgements

The authors would like to thank Ashwin Sridharan from Sprint Labs for providing data about packet size distributions.

References

1. Bormann, C., Burmeister, C., Degermark, M., Fukushima, H., Hannu, H., Jonsson, L.-E., Hakenberg, R., Koren, T., Le, K., Liu, Z., Martensson, A., Miyazaki, A., Svanbro, K., Wiebke, T., Yoshimura, T., Zheng, H.: RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed. RFC 3095 (Proposed Standard) (July 2001)
2. CAIDA. Packet Length Distributions (August 2004), `http://www.caida.org/analysis/AIX/plen_hist`
3. Casner, S., Jacobson, V.: Compressing IP/UDP/RTP Headers for Low-Speed Serial Links. RFC 2508 (Proposed Standard) (February 1999)
4. Coulouris, G.F., Dollimore, J., Kindberg, T.: Distributed Systems: Concepts and Design, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (2001)
5. Degermark, M., Nordgren, B., Pink, S.: IP Header Compression. RFC 2507 (Proposed Standard) (February 1999)
6. Fischer, M., Lynch, N., Paterson, M.: Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM 32(2), 374–382 (1985)
7. IEEE. IEEE 802.11 Optimal Performances: RTS/CTS Mechanism vs. Basic Access. In: Proceedings of PIMRC (2002)
8. Jacobson, V.: Compressing TCP/IP Headers for Low-Speed Serial Links. RFC 1144 (Proposed Standard) (February 1990)
9. The VINT Project. The ns Manual. Available in html, postscript and PDF (December 2003)
10. Sprint. IP Monitoring Project (February 2004), `http://ipmon.sprint.com/packstat/packetoverview.php`