Stable ping over 802.11 WiFi

J Rick Ramstetter
jramstet@cs.rutgers.edu

Abstract:
The ability to provide for the localization and ranging of moving devices using 802.11 WiFi and other consumer grade equipment is sought, in part due to the low cost and large user base of such equipment. A primary obstacle of efforts in this direction is the high variance and high average magnitude seen in echo/response round trip times (ICMP "Ping," for example) over 802.11 equipment. These problems stem in large part from the deferred processing paradigm of modern device drivers. This work demonstrates that by violating this paradigm 1) a reduction of the average round trip time magnitude for echoes and responses can be had and 2) a reduction in corresponding variance can be had. This work further demonstrates the ability of consumer grade equipment to more precisely record the departure and arrival time of 802.11 packets.

1. Introduction and background

With the knowledge of total time taken for wave to propagate to some destination and back (a round trip time, or RTT), it is possible to estimate the distance of that destination from the source. Recall from introductory physics that *distance = time * speed.* For an unmoving source and destination the round trip time would be :

*Round trip time = (2*distance) / speed*

From this, it can be expected that an increase in distance would result in a similar increase in round trip time. For example, the RTT in Figure 1's (B) should increase from the RTT of (A) by a factor involving two times the increase in distance between source and destination.

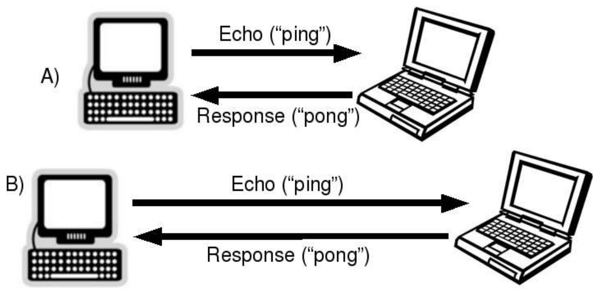


Figure 1: The RTT of (A) should be less than that of (B)

Unfortunately such is not generally the case. Utilizing consumer grade, 802.11 equipment in addition to a consumer grade hardware platform generally dictates that the destination node must process incoming packets in a "deferred processing" fashion.

To illustrate, let node *X* be a consumer grade laptop with an 802.11 WiFi card using GNU/Linux. Four tasks must be carried out when a packet arrives at *X* over 802.11 WiFi:
(0) store the packet
(1) analyze the packet
(2) form an appropriate response
(3) send the response

Say *X* receives an ICMP Echo ("ping") request over 802.11 WiFi at time *t. X* must store the packet at time *t* (task (0) above) otherwise the packet would be lost. However, *X* need not do any of (1), (2), or (3) (from above) at time *t.* Indeed, if *X* was doing something else of high importance at time *t* (say, writing data to a hard drive), it is quite likely that (1), (2) and (3) *will not* happen at time *t.*

The above illustration assumes (and most operating systems assume) that on node *X* the 802.11 WiFi device's hardware driver is well behaved and implements some type "deferred processing." Supposedly, the driver will do only the minimal amount of work as is required when the packet arrives at time *t,* thus allowing the operating system to more intelligently schedule processing time. When the operating system deems itself ready at some time *t+i*, it will return processing to the device driver to carry out tasks (1), (2) and (3). For most purposes this deferred processing scheme is advantageous. For the purposes of localization and ranging it is strictly harmful.

The work presented subsequently in this paper will demonstrate that by violating the paradigm of deferred processing in a specific 802.11 device driver, substantial reductions in the variance of echo / response RTTs can be had. This work also demonstrates the ability of consumer grade equipment to measure the arrival and departure time of packets with finer granularity than most utilities ("/bin/ping," for example) offer. The impact of these changes on other aspects of computer usage is not explored as this work seeks only to prove the feasibility of a stable and minimized RTT over 802.11 WiFi.

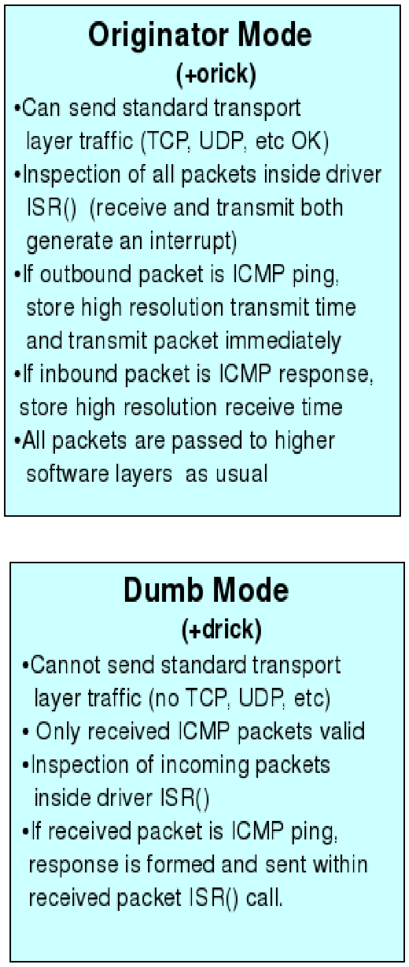
2. Theory of operation

The general idea is to accelerate the priority of incoming and outbound ICMP packets, and record high resolution timestamps for those packets when necessary. The problems involved with optimizing round trip time can be illustrated as follows. Node *X* wishes to gauge its distance to Node *Y*. If *X* queues a packet for transmit to *Y* at time *t*, the time spent waiting in the transmit queue on *X* is irrelevant provided that *X* stores the physical transmit time (say, *t+timeWaitingInQueue*). Let *X* receive a response from *Y* at time *tR*. Clearly, a rough gauge of the distance between *X* and *Y* can be had from

*distance = 1/2 * (Round trip time * speed) = 1/2 * (t + timeWaitingInQueue - tR)* speed*

For this paper's purposes, we seek to minimize *X*'s *timeWaitingInQueue* when possible, though such is not necessary. However, of utmost necessity is a minimization of queue times on *Y*, as *X* cannot take into account delays introduced by *Y*. Node *X* cannot in real time know the value of *Y*'s *timeWaitingInQueue.*

Thus we desire two modes of operation: 1) an "Originator" mode which records high resolution timestamps for inbound and outbound packets and 2) a "Dumb" node which processes and responds to an inbound ICMP packet as quickly as is possible. More details regrading these introduced modes can be found in Figures 2 & 3.



Originator Mode
(+orick)
- Can send standard transport layer traffic (TCP, UDP, etc OK)
- Inspection of all packets inside driver ISR() (receive and transmit both generate an interrupt)
- If outbound packet is ICMP ping, store high resolution transmit time and transmit packet immediately
- If inbound packet is ICMP response, store high resolution receive time
- All packets are passed to higher software layers as usual

Dumb Mode
(+drick)
- Cannot send standard transport layer traffic (no TCP, UDP, etc)
- Only received ICMP packets valid
- Inspection of incoming packets inside driver ISR()
- If received packet is ICMP ping, response is formed and sent within received packet ISR() call.

Figure 2: Detailed explanation of introduced modes

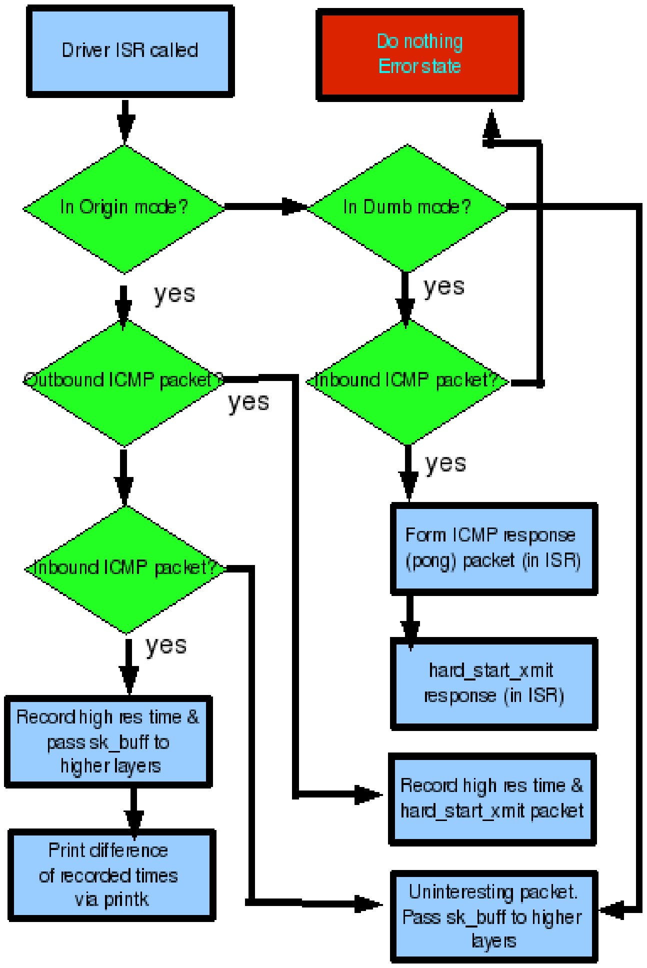

Driver ISR called
Do nothing
Error state
In Origin mode?
In Dumb mode?
yes
yes
Outbound ICMP packet?
yes
Inbound ICMP packet?
yes
Form ICMP response (pong) packet (in ISR)
Inbound ICMP packet?
yes
hard_start_xmit response (in ISR)
Record high res time & pass sk_buff to higher layers
Record high res time & hard_start_xmit packet
Print difference of recorded times via printk
Uninteresting packet. Pass sk_buff to higher layers

Figure 3: Flow chart of execution inside modified driver ISR

3. Implementation details

3.1 Hardware Base System

This work strove to utilize as much consumer grade equipment and software as was possible. For the hardware platform, only the X86 cpu architecture as used. Two machines of high processing power (>1GHz) were used. One machine of lower (<150MHz) processing power was used. A Soekris embedded device based on the AMD Geode X86 processor family was also used. The operating used was GNU/Linux, using version 2.6.24 of the kernel. All machines, except the Soekris, where using a X11 + Gnome GUI during testing. The Soekris used an older kernel version. This work assumes that the GNU/Linux can be considered "consumer grade."

3.2 The 802.11 Device

The Atheros 5212 WiFi chip was chosen as the target device primarily for its compatibility with the Madwifi-project.org device drivers. These drivers are open source, well written, and easily extensible. These drivers further provide significant and easily utilized debugging mechanisms. As an added bonus, an on-campus source had multiple Atheros 5212 based WiFi cards available at the onset of this work.

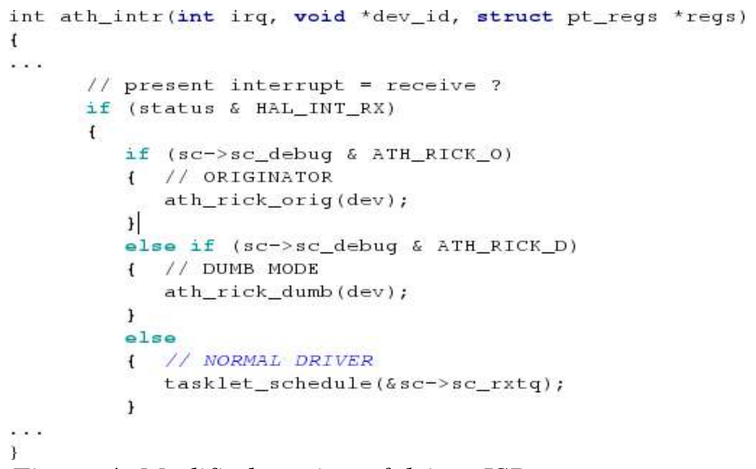
A downside to the Madwifi drivers (explained later) is their use of a closed source hardware abstraction layer of questionable performance.

The decision to make use of ICMP ping and pong packets (versus, say, introducing a custom 802.11 frame) was made due to a lack of robustness in consumer grade WiFi equipment. For example, it was observed that many models of consumer grade wireless routers crashed upon receipt of a malformed 802.11 frame. Likewise, certain brands of wireless cards become unresponsive after receipt of such a frame.

3.2 Specific Modifications

A working knowledge of the Linux Kernel is henceforth assumed.

The Madwifi drivers create (at driver load) a variety of kernel system controls (sysctls) under */proc/sys/dev/ath*. Of these many sysctls, *debug* is most relevant to this work. This debug sysctl is a bitmask that can be easily read by the Madwifi driver. The current bitmask can be altered via the userspace utility "athdebug" (which is included with the Madwifi drivers) or by echoing and redirecting appropriate values to */proc/sys/dev/ath/debug*. The flow of the Madwifi drivers is altered according to the currently set bitmask. For example, "athdebug +xmit" (corresponding to a bitmask of 0x00000001) results in outbound packets being logged to the system log.



```
int ath_intr(int irq, void *dev_id, struct pt_regs *regs)
{
...
        // present interrupt = receive ?
        if (status & HAL_INT_RX)
        {
            if (sc->sc_debug & ATH_RICK_O)
            {   // ORIGINATOR
                ath_rick_orig(dev);
            }
            else if (sc->sc_debug & ATH_RICK_D)
            {   // DUMB MODE
                ath_rick_dumb(dev);
            }
            else
            {   // NORMAL DRIVER
                tasklet_schedule(&sc->sc_rxtq);
            }
...
}
```

Figure 4: Modified version of driver ISR

This work has introduced two new modes, +orick for Originator Mode (corresponding 0x02000000) and +drick for Dumb Node Mode (corresponding to 0x02000000). Once the driver ISR has determined the present interrupt was caused by a packet being received, it checks the current debug bitmask to see if either of Originator or Dumb modes are enabled. If so, appropriate actions are taken. General execution flow inside the ISR is illustrated by Figures 3 and 4.

As seen in Figure 4, this work's modifications to the driver avoids the use of driver bottom halves (specifically, tasklets). The functions ath_rick_orig and ath_rick_dumb, then, must provide the necessary parts of the functionality normally implemented in the driver bottom half.

3.2.1 Specifics of ath_rick_dumb (Dumb mode)

The function *ath_rick_dumb* is a heavily modified version of the driver's *ath_rx_tasklet*, the function added to the kernel tasklet queue by *tasklet_schedule(&sc->sc_rxtq).* This modified version strips out much functionality; for example, 802.11 monitor mode and various sanity checks against the received packet are both removed.

Say a packet is received at Dumb mode Node *X*. The ISR of *X*'s Madwifi driver calls *ath_rick_dumb*. The 802.11 headers are removed from the received packet, at which point the packet is verified to be of ICMP type by header inspection. If yes, code from the Linux Kernel's icmp.c is used to verify that the current packet was destined to the *X*. If so, a response is formed (again using icmp.c code) and immediately transmitted.

This immediate transmission takes place by a call to the *hard_start_xmit* function for the driver, which is *ath_hardstart*. Immediately prior to the *ath_hardstart* call, *skb_queue_purge* and *netif_stop_queue* are both called. These ensure that the operating system cannot attempt to transmit a packet simultaneously to the current transmission. Once the *ath_hardstart* function returns, *netif_start_queue* is called to re-enable the interface.

This implementation leaves much room for improvement. The device operating in Dumb mode appears to the operating system as a normal networking device. This is unfortunate as the device can only 1) receive ICMP packets and 2) respond to them. GNU/Linux sees the device and can attempt to pass traffic over it, but these attempts are guaranteed to fail.

3.2.2 Specifics of ath_rick_orig (Originator mode)

An immediate clarification is in order. Originator mode encompasses two functions: *ath_rick_orig* handles receives in originator mode, while *ath_rick_orig_tx* handles transmits.

The function *ath_rick_orig_tx* works as follows. Say an interrupt is generated and the ISR is called. If the ISR determines that the interrupt was caused by an outbound packet, and further that Originator mode is enabled, the driver calls *ath_rick_orig_tx*. This method is nearly identical to the transmit tasklet function normally used by the driver, *ath_tx_tasklet.* However, this method is not added to a kernel tasklet queue but rather called immediately.

The primary difference in *ath_rick_orig_tx* from *ath_tx_tasklet* is stateful inspection of outbound packets to check for ICMP pings. If an outbound ICMP ping is seen, a high resolution transmission time is recorded and *hard_start_xmit* is called on the packet's buffer. Similar to *ath_rick_dumb*, this call to *hard_start_xmit* requires that kernel networking queues be flushed and stopped.

The function *ath_rick_orig* is based on *ath_rx_tasklet* and is somewhat similar to *ath_rick_dumb* (see Section 3.2.1). The relative opposite of *ath_rick_orig_tx*, it is called when the ISR determines that a packet has been received. The driver temporarily records a high resolution inbound time. If the received packet is of type ICMP ping response ("pong"), this high resolution inbound time is used to determine the total RTT (see Section 3.2.4). By avoiding the use of kernel tasklets (that is, avoiding the use of *ath_rx_tasklet*), the priority of the incoming packet is boosted.

Room for improvement is left in the recording of timestamps. At present, they are recorded only after some minimal processing has taken place and the type of an arriving or departing packet has been ascertained. Ideally, the 802.11 device would record a timestamp (at the 802.11 chipset level) then pass this timestamp to the driver when an ICMP packet has been detected (or is known to have been transmitted).

3.2.3 Specifics of high resolution timestamps (Originator mode)

The Linux kernel header file time.h's timeval struct is being used to store times. This struct contains two members:

```
time_t          tv_sec      //seconds
suseconds_t     tv_usec     //microseconds since the last second tick
```

For this work's purposes, the struct is stored in the Madwifi driver's softc. For reference, a softc is a driver specific struct that contains a mesh of data (for example, a list of associated hardware devices). The Madwifi softc has been modified to include a circular array of timeval structs store the most recent outbound transmit times.

The values of these struct members (*tv_sec* and t*v_usec*) are set by a call to time.h's *gettimeofday* function. This function is called inside the scope of the ISR (specifically, inside a*th_rick_orig_tx*) directly before a packet is transmitted or passed to higher networking layers.

Note that the receive timestamp for inbound packets need not be stored outside the scope of the current ISR call. Thus, only timestamps for outbound packets are stored in the Madwifi driver's softc. This is because it can be guaranteed that only one packet will be received at a time, and that calls to the ISR will not be interrupted by subsequent calls to the ISR (both thanks to hardware buffers on the Atheros 5212 device).

At every ICMP response ("pong") packet receive (every call to ath_rick_orig), an RTT time is computed using the receive time for an incoming packet and the earliest entry in the circular array (assumed to be the associated transmit time). See section 3.2.4 for details regarding error mitigation. The computed RTT time is printed to the user via the Linux syslog daemon (specifically, a call to *printk* is made).

The circular array is admittedly less than ideal. Care must be taken to avoid the dropping of data in the array (as implemented, the array will discard timestamps if it becomes "too full"). If such corruption does occur, the user is notified via a message in the system log. In testing, setting a threshold of at least .2 (2/10) seconds between outbound ICMP ping packets resulted in no data corruption. As can be expected, pinging a non-existent or non-responsive host will fill the circular array: outbound timestamps are being added to the array at every transmit, but no receives are happening (and thus nothing is removed from the array).

3.2.4 Error mitigation (Originator mode)

If the computed RTT at a receive is above some threshold, it is recomputed using the next outbound timestamp from the circular array. This would occur, for example, if the (relative) entry 0 of the outbound timestamp array corresponds to a packet that has long since been dropped .

Additionally, the timestamp of the most recently received pong is stored in the driver softc. This value is overwritten every time a pong is received and is not used to compute the RTT. Rather, this value serves as an elapsed time threshold. At every transmit (inside *ath_rick_orig_tx*), if some threshold of time has passed since the last received pong, the packets relevant to all entries in the outbound timestamp circular array are assumed to have been dropped, and the circular array is flushed.

4. Results

4.1 1000 pings between two devices of high processing power (>1GHz CPU).

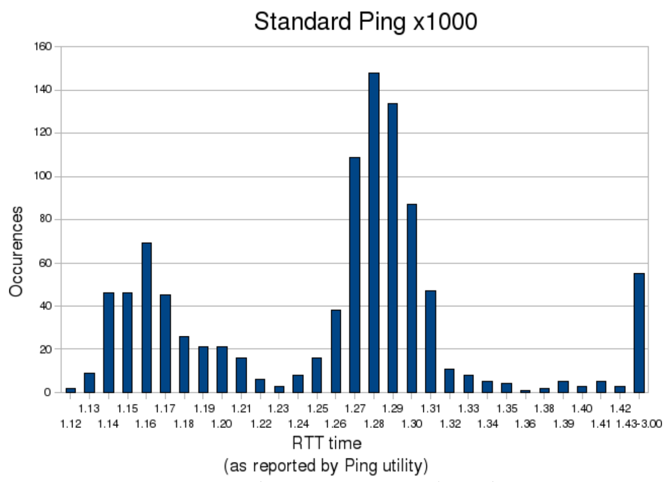


Figure 5: 1000 pings between two unaltered, unmoving Madwifi/Atheros devices.

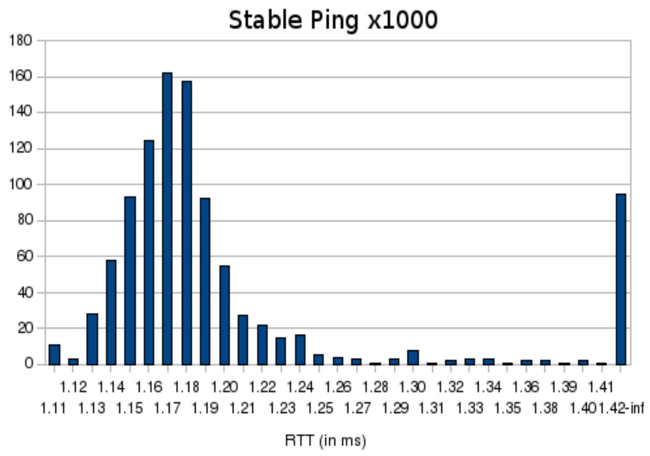


Figure 6: 1000 pings between an Originator mode Madwifi/Atheros device and a Dumb mode Madwifi/Atheros device. Both devices unmoving. Times reported by Ping utility

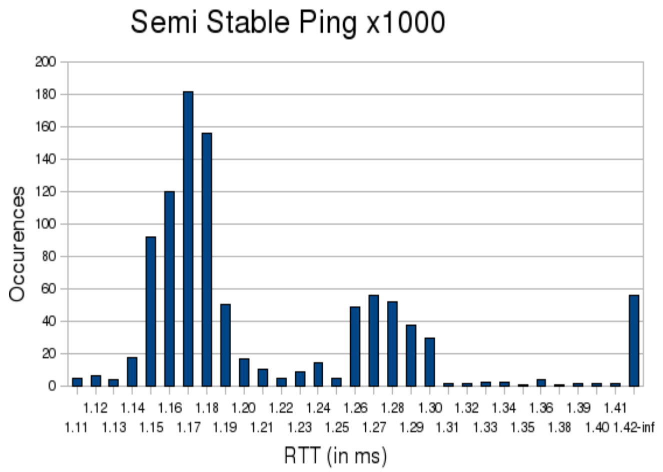


Figure 7: 1000 pings between an Originator mode Madwifi/Atheros device and an unaltered Madwifi/Atheros device. Both devices unmoving. Times reported by Ping utility.

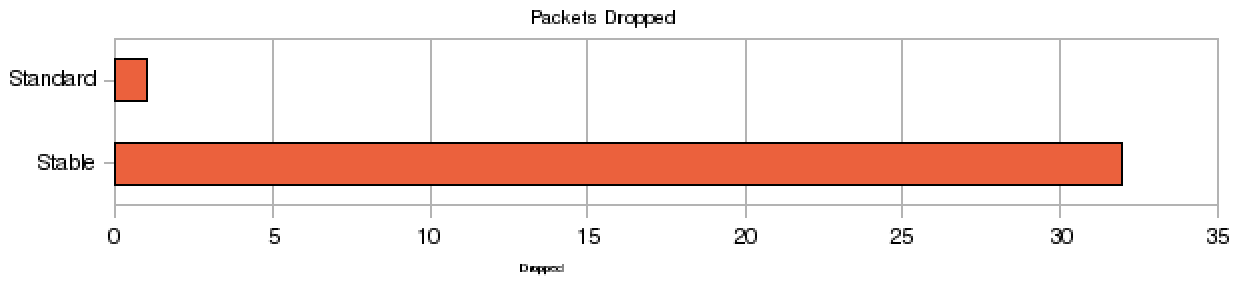


Figure 8: Packets dropped during 1000 pings: Standard vs Stable (semi-stable not plotted)

4. Results (continued)

4.2 1000 pings from a device of high processing power (>1GHz CPU) to a device of low processing power (<150MHz CPU)

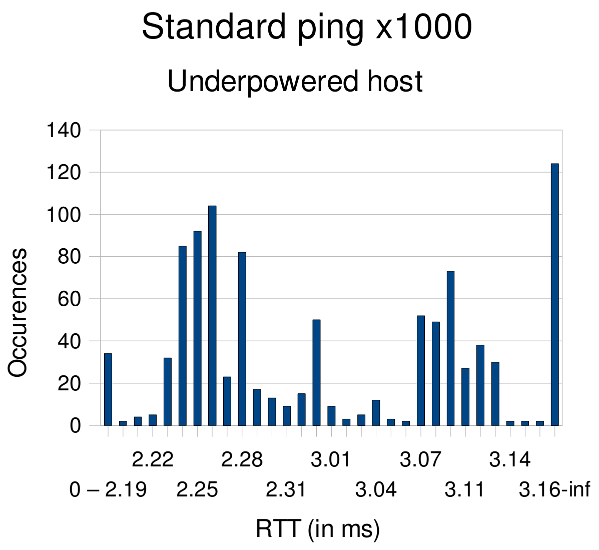


Figure 10: 1000 pings from an Atheros/Madwifi device of high processing power to a similar device of low processing power.

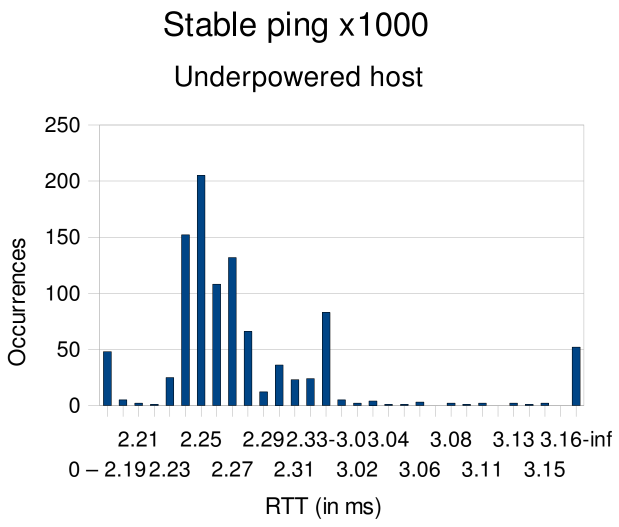


Figure 9: 1000 stable pings from an Atheros/Madwifi device of high processing power to a similar device of low processing power.

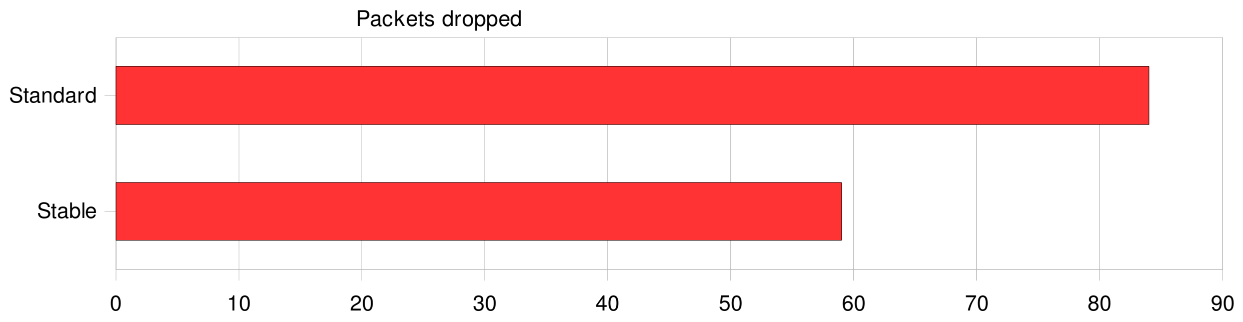
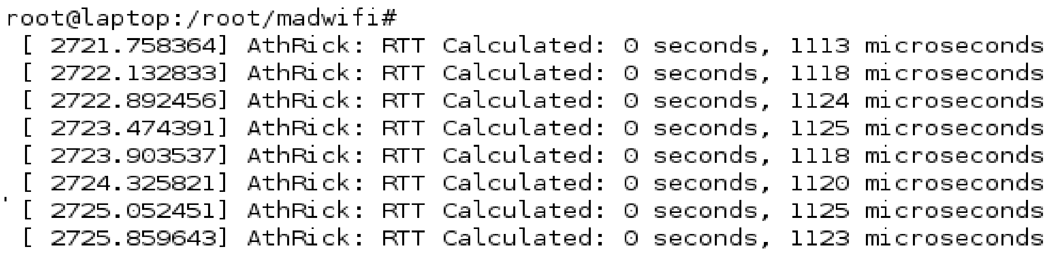


Figure 11: Packets dropped during 1000 pings to an underpowered host. Standard vs stable.

4.3 High resolution timestamps



```
root@laptop:/root/madwifi#
 [ 2721.758364] AthRick: RTT Calculated: 0 seconds, 1113 microseconds
 [ 2722.132833] AthRick: RTT Calculated: 0 seconds, 1118 microseconds
 [ 2722.892456] AthRick: RTT Calculated: 0 seconds, 1124 microseconds
 [ 2723.474391] AthRick: RTT Calculated: 0 seconds, 1125 microseconds
 [ 2723.903537] AthRick: RTT Calculated: 0 seconds, 1118 microseconds
 [ 2724.325821] AthRick: RTT Calculated: 0 seconds, 1120 microseconds
 [ 2725.052451] AthRick: RTT Calculated: 0 seconds, 1125 microseconds
 [ 2725.859643] AthRick: RTT Calculated: 0 seconds, 1123 microseconds
```

Figure 12: Screenshot of driver output appearing to root user console via Syslog KERN_EMERG printk calls. Note: /bin/ping is running in another console as non-superuser

5. Analysis & Conclusions

The results presented indicate that the modifications to the Madwifi driver have been successful in reducing both the magnitude and variance of round trip times over 802.11. This has been accomplished by eliminating the driver's use of deferred processing and introducing application specific optimizations where possible.

The most pronounced change has occurred in the round trip time between a high powered (>1GHz) and low powered (<150MHz) computer. Given that the background processing load on such an underpowered system is usually a high percentage of total processing power (at least in a GUI), this makes sense.

Purists may argue that the changes implemented in this work violate the OSI 7 Layer model of networking. This argument is correct but fails to recall the highly specific nature of this work. It is extremely unlikely that such changes would ever be implemented in the actual Madwifi driver as the changes are application specific.

Further testing and refinement of the high resolution printing mechanisms is needed. At present, the mechanisms do not well correlate with their userspace counterparts (say, the output of the Ping utility). Indeed, the results of the two outputs can vary by as much as .2 (2/10) milliseconds. The reasons for this remain unclear, though it seems highly unlikely this large difference can be accounted for by an increase in resolution.

The specific modifications used in this work should be be translatable (at least at a conceptual level) to other, open source 802.11 drivers. To rephrase, no functionality specific to the Madwifi drivers has been used.

*****

The author would like to thank the subscribers of the Madwifi-Devel mailing list for their support, and Dr. Rich Martin for providing much of the equipment used.