Realtime Multicast for SDR Module Interconnection

Phil Karn, KA9Q
karn@ka9q.net

Abstract

I have developed a set of software defined radio receiver modules to demonstrate the utility of standard Internet real-time and multicast protocols for interprocess communication. The multicast streams convey complex digital IF sample streams, uncompressed and compressed audio and decoded physical layer frames (e.g., AX.25). One module can feed any number of others, and it is easy to move and restart individual modules without restarting the entire system. The package is currently being used for general analog reception and to process balloon APRS tracking transmissions but has many other applications including satellite operations and digital voice. The source code is in C, is available as open source[1] and runs on any UNIX-like operating system, including Mac OSX and Linux on Intel/AMD x86 and ARM (e.g., Raspberry Pi) systems.

Key words

Software defined radio, multicasting, real time protocol, Internet

RTP background

TCP, the standard Internet connection-oriented stream transport protocol, is excellent at what it does but it is ill-suited to real time and multicast applications. TCP's use of acknowledgements and retransmissions introduces variable and possibly unbounded end-to-end delays intolerable in real-time communications, and it is usually impractical to have every receiver in a large multicast group return its own acknowledgements.

Hence the *Real Time Protocol* (RTP) was developed. Unlike TCP, RTP runs over the User Datagram Protocol (UDP) rather than directly above IP. But RTP is functionally a transport protocol so it provides some of the same features as TCP (except acknowledgements). Sequence numbering informs the receiver of missing or reordered packets. Timestamps maintain time synchronization when packets are lost or deliberately not sent, e.g., with variable-rate compression schemes and discontinuous (PTT/VOX) transmission. Other fields distinguish sources, identify upstream contributors and give the format and type of data being sent.

The core parts of RTP and ancillary protocols are defined in Internet RFC 3550[2], published in 2003.

IP multicasting

RTP is ubiquitous in voice-over-IP telephony (VoIP) but it was designed for much more. Although VoIP is usually sent over ordinary unicast point-to-point Internet paths, its origins are in IP multicasting: the efficient delivery of packets to a set of recipients that

[1] https://github.com/ka9q/ka9q-radio

[2] http://www.rfc-base.org/rfc-3550.html

may be large and time-varying. One obvious application is telephone conference calling; another is one-way broadcasting to limited audiences. AT&T's U-verse service[3] uses multicast RTP to distribute TV programming over DSL and other relatively slow data links unable to carry a full composite channel set as in conventional cable TV. The same technology is also deployed in other countries.[4]

Ordinary unicast Internet packets are delivered to only one destination address, at most. If you have multiple recipients, you must send each its own unicast packet. This is often done, but there's a more efficient way.

IPv4 addresses 224.0.0.0 to 239.255.255.255 are *multicast* group addresses that appear only in the IP destination field (never the source). Packets sent to a multicast address are duplicated by multicast routers as necessary to efficiently deliver one copy to each member of the group. End points use the *Internet Group Management Protocol* (IGMP) to join and leave individual groups; in UNIX and Linux, these messages are automatically sent by the kernel when an application joins a group, leaves or terminates.

Multicasting is even older than RTP; RFC1112[5], published in 1989, defines the core concepts and procedures. IGMP has gone through three revisions, most recently in RFCs 3376[6] and 4604[7] published in 2002 and 2006. IPv6 has its own group management protocol called Multicast Listener Discovery.[8] Most subsequent work has been in multicast routing, which doesn't affect how host computers and applications use it.

IP multicasting is ubiquitous on LANs for local resource discovery, especially by smart phones, tablets, printers and entertainment systems. But it is little used for media streaming outside “walled gardens" such as AT&T Uverse and some corporate and university networks. It never caught on over the public Internet, which seems a shame since so much Internet content (from public websites to software updates) is inherently broadcast or multicast in nature. Apparently fibers (and CDNs -- content distribution networks) are now so cheap that providers don't mind the inherent inefficiencies of sending the same data over and over.

RTP/multicast for SDR applications

Of course you can still use these protocols in the privacy of your own home. (They can also be tunneled over the public Internet; more about this later.) It occurred to me that RTP over IP multicasting might be well suited to interconnecting software-defined radio components running on arbitrary computers around my house. Except when specific hardware is required, any module can run on any computer with enough spare cycles. One sender can trivially feed several listeners at the same time without any special

[3] https://en.wikipedia.org/wiki/AT%26T_U-verse

[4] https://en.wikipedia.org/wiki/IPTV

[5] https://www.rfc-editor.org/info/rfc1112

[6] https://www.rfc-editor.org/info/rfc3376

[7] https://www.rfc-editor.org/info/rfc4604

[8] https://www.rfc-editor.org/info/rfc3810

arrangements. (A source doesn't even have to know how many listeners it has.) A source simply transmits to a group and listeners join and leave it at will. My Ethernet switches and routers do the necessary packet copying and forwarding.

Individual SDR modules can be started, stopped, and replaced as necessary without having to restart every other module in the signal path as would be necessary if they were connected with UNIX pipelines (on the same system) or TCP network connections.

Because networking is always more expensive than a simple procedure call inside a program, it wouldn't make sense to use multicast RTP between *every* conceivable signal processing block. I reserve it for well defined points where the versatility of modularity justifies the cost. So far I have created the following types of multicast streams:

1. Complex (I/Q) digital IF signals (with metadata)
2. Uncompressed PCM audio (mono or stereo)
3. Low bit rate lossy compressed audio (mono or stereo)
4. Raw demodulated physical layer data frames (e.g., AX.25).

Utilities also exist to record and play back each stream type for backup and testing purposes.

Here I describe the SDR modules I've written to demonstrate the RTP/multicast concept, along with their interface formats.

Digital IF streams

These convey I/Q (complex) samples from various SDR front ends to programs that receive, demodulate or record them. I've found it convenient to host SDR hardware on a dedicated Raspberry Pi or small x86 computer that multicasts the IF sample data onto Ethernet. The radio hardware can then be placed at the antenna, allowing an RF feed line to be replaced with Ethernet over twisted pair or optical fiber.

The data usually consists of interleaved 16-bit integer samples. I used 8-bit integers with the HackRF One[9] (which has 8-bit A/Ds) but returned to 16 bits to handle the increased dynamic range after optional decimation. I'm considering the new IEEE standard half-precision (16 bit) floating point format[10] for more dynamic range without increasing the data rate.

Each digital IF packet includes standard Ethernet, IP, UDP and RTP headers plus a custom metadata header with the time of day (nanosecond resolution), A/D sample rate, signal frequency (i.e., the radio frequency corresponding to 0 Hz in the I/Q stream), and the analog gains ahead of the A/D.[11]

This being a custom format, I saw no need to follow the usual Internet conventions on byte ordering (beyond the standard IP/UDP/RTP headers, which are always big-endian). For efficiency, A/D samples and metadata are little-endian, as are the x86, ARM and

[9] https://greatscottgadgets.com/hackrf/

[10] https://en.wikipedia.org/wiki/Half-precision_floating-point_format

[11] I briefly considered VITA-49, but it reminded me of OSI: too general, complex and vague. I'm willing to reconsider, however.

nearly every other surviving computer architecture.[12]

Tuning and gain setting commands are unicast to the digital IF source, which then sends them to the hardware. The command format closely resembles the metadata header. In principle, commands could be multicast to the same group as the data stream (with a distinguishing port or type value) so that everyone could see them, but this information is already echoed in the metadata. No special configuration is necessary, as the commands are sent to the source address of the IF stream with the UDP destination port being the UDP source port of the IF stream plus one. This also works for multiple streams from the same computer.

At the moment there is no command authentication or contention resolution; more about this later.

Digital IF stream generators and consumers

Three programs currently produce digital IF multicast streams: *funcube, hackrf* and *iqplay.*

The first two read from the AMSAT UK Funcube Dongle Pro+[13] and HackRF One, respectively. *iqplay* plays back locally generated or previously recorded streams for test purposes as though they originated from *funcube* or *hackrf.*

Two programs consume I/Q streams: *iqrecord* and *radio*. The first is the record counterpart to the play program. As an experiment, *iqrecord* stores metadata in extended file attributes;[14] the files themselves contain only headerless I/Q sample data. This may or may not turn out to be a wise design.

The ability to record raw IF signals independently of processing them is very useful during critical spacecraft or balloon mission events. If something happens to the hardware or software processing signals in real time, the signal is still preserved in a recording that can be replayed later.

The *radio* program

This is currently the largest and most complex processing element, with an elaborate user interface. It implements a full-blown narrowband general coverage multimode receiver that processes the digital IF stream from the SDR front end and produces digita audio. The user interface uses the ncurses package for efficiency over slow network connections.[15]

There is as yet no waterfall display; for modularity this should be a separate dedicated program that reads the same digital IF stream.

All processing is in the complex domain wit the input passband centered at 0 Hz, corresponding to the radio frequency set by the tuner in the SDR front end. Fine tuning (e.g. between -96 kHz and +96 kHz for a 192 kHz complex sample stream) is with a software

[12] https://en.wikipedia.org/wiki/Endianness

[13] http://www.funcubedongle.com/

[14] https://en.wikipedia.org/wiki/Extended_file_attributes

[15] https://en.wikipedia.org/wiki/Ncurses

oscillator implemented in double precision floating point for essentially perfect frequency resolution. When the edge of the I/Q passband is approached, a command is automatically sent to retune the SDR front end. (The edges of the passband are avoided as they are likely to contain aliased signal components.)

The user may opt to shift the first (hardware) and second (software) LOs in tandem to maintain a constant radio frequency. This is useful for identifying possible images (from inadequate I/Q gain/phase correction) or aliases (from imperfect filtering) and shifting them away from the desired signal. In principle this could be done automatically by finding the offsets that minimize the detector output, on the principle that images or aliases can only increase the detected output.

Optional Doppler steering for satellites is provided by reading a UNIX pipe from a separate program supplying timestamped relative velocity information from an open-loop orbit model. The downconverter interpolates between velocity points to maintain continuous phase and frequency; this should assist coherent demodulators (e.g., for BPSK) that must track carrier phase. To avoid front-end retuning transients, it should be set before a pass to keep the signal entirely within the digital IF bandwidth.

This system kept the CW beacon from an old 70 cm Japanese cubesat in a 400 Hz filter from AOS to LOS with no manual intervention and no audible retuning events. This required a current set of Keplerian orbital elements and accurate system time controlled by the Network Time Protocol (NTP).

Pre-detection filtering is by fast convolution, which also reduces the sample rate to 48 kHz. The user can independently adjust the filter skirts and Kaiser window[16] parameter (which affects the skirt shape). There's also a post-detection passband shift mainly useful for narrow modes like CW or PSK31. The filter block size (default 20 ms) can be changed with a program restart. Because the filter is complex, asymmetric filters for SSB are easily implemented without an explicit Hilbert transform.

Signal and noise amplitudes are independently measured so that pre-detection SNR can be calculated. Noise spectral density (N0) is estimated by first averaging the power density outside the signal passband on the assumption that it contains mostly or only noise. Then a new average is computed, excluding frequency bins with energies above some threshold relative to the first average on the assumption that they contain other signals, not just noise. Although not foolproof, this algorithm works remarkably well with just a few passes.

Signal demodulators

Three demodulators for analog modes are currently provided. The *linear* demodulator handles SSB, CW, coherent AM, DSB, ISB (independent sideband) and similar modes. A software PLL recovers the carrier in the coherent AM and DSB modes (in DSB, the carrier is first regenerated by squaring). This is especially useful for frequency calibration to a known source.

The *AM* demodulator is a simple envelope detector.

[16] https://en.wikipedia.org/wiki/Kaiser_window

The *FM* demodulator uses direct arctangent calculation.[17] You can listen to either the raw FM output or to a de-emphasized and PL-filtered version[18]. A FFT determines the frequency of any PL tone to 0.1 Hz precision.

The FM demodulator uses an experimental threshold extension algorithm of my own design. Many FM demodulators first limit the signal to a constant amplitude, but this discards potentially useful information. I use sample amplitudes as reliability hints akin to "soft decision" Viterbi FEC decoding. A weak sample is more likely to be part of a “click" than a strong one, so I simply blank them. I found empirically that blanking samples with amplitudes below about 0.55 times average produces good results: much less "popcorn" noise than pure FM detection, without much loss of high audio frequencies. I haven’t compared it with the usual methods of threshold extension such as a tracking PLL.

The FM squelch operates from first principles; I measure the amplitude and the variance of the pre-detection FM signal and calculate its SNR. If it is above a fixed threshold (+3 dB), the squelch opens. This works so well I've felt no need for a manual “squelch knob". Originally the squelch closed in complete silence, but I had to delay it by one 20ms filter frame to ensure filter flushing when demodulating AX.25 packets; unfortunately this reintroduced a brief squelch tail.

PCM audio

The *radio* program produces a 16-bit 48 kHz PCM mono or stereo audio multicast stream, depending on demodulator mode. I use an existing Internet convention for uncompressed PCM in RTP with big-endian samples and different payload types for mono and stereo. This is fully compatible with Internet audio players such as *vlc*[19].

PCM audio streams can also be heard with the *monitor* program, recorded with the *iqrecord* program (which automatically detects IF and audio streams) or further processed by other programs, e.g., the *opus* lossy audio compressor or the *packet* AFSK/AX.25 demodulator. It should be easy to adapt other signal processing programs, e.g., WSJT-X, to accept PCM audio multicast streams. This would avoid any need to use a computer’s sound subsystem.

Opus compressed audio

Mono 16-bit PCM at 48 kHz consumes 768 kb/s; fine on a wired Ethernet but uncomfortably high even on WiFi. This high rate is overkill for communications-quality audio, but rather than reduce the sample rate I added the Opus general-purpose lossy audio codec. It is intended only for listening; digital demodulators should always process the PCM stream.

Opus is an IETF standard described in RFC 6716 and elsewhere[20]. An excellent refer-

The DSP literature is full of articles on fast arctangent approximations for FM demodulation but even on the Raspberry Pi, the regular arctangent library function is plenty fast. Hardware has come a long way!

[18] I suppose I should say CTCSS, but it's a mouthful.

[19] http://www.videolan.org

[20] https://www.rfc-editor.org/info/rfc6716

ence implementation is available in open source, [21] and unlike many other codecs no license fees are required. It combines several algorithms to span a very wide range of compressed bit rates, from 6 kb/s to 510 kb/s. To my ear 32 kb/s is plenty for communications quality speech and lower data rates (down to 6 kb/s) are quite acceptable.

Opus strongly prefers a constant 48 kHz sample rate even for communications quality speech, and there is no bit rate advantage to lowering the sample rate or to operating the codec in mono, as opposed to stereo with the same signal in each channel. I think it's an excellent, easy-to-use codec; the price is certainly right. Along with CODEC2[22] for very low bit rates I'd really like to see Opus widely used in ham radio. We've been hamstrung by proprietary codecs far too long!

The program *opus* listens to PCM on one multicast group and repeats it in Opus-compressed form on another. Options select the input and output multicast groups, the target compressed bit rate (default 32 kb/s) and the codec block duration (default 20 ms). It also enables discontinuous transmission, a VOX-like mechanism that reduces the output rate to about 3 (short) packets per second when there is (near) silence on the PCM input. For FM reception the 'radio' program stops the PCM stream when the squelch closes, so the Opus stream also stops even if discontinuous mode is not selected.

Because the PCM and Opus streams are on separate multicast groups, an audio player can simply select the appropriate group according to the available network capacity. Opus is especially useful for remote monitoring.

The *monitor* program

The *monitor* program mentioned above can play both Opus and PCM streams. It can handle any number of streams on any number of multicast groups, and the user interface lets the user adjust the gain and stereo position of each stream to support a “virtual conference table" feature where each participant has his own place in the stereo image. Stereo positioning is performed by varying the relative amplitudes and delays of the two channels.

The *monitor* program is conceptually simple, but it probably gave me more headaches than anything else I wrote. I had to overcome WiFi timing jitter and several bugs in audio sound libraries and device drivers. I became quite familiar with how WiFi access points handle multicast traffic; more on this later.

Handling jitter can be difficult, especially when you want to minimize latency. There seem to be two common approaches to managing a playout buffer: maintaining a fixed delay and simply dropping any packets that arrive too late; or increasing the delay as needed when packets arrive late. I tried both and chose the latter, but this creates the problem of keeping the delay from becoming excessive.

The FM squelch helps a lot. Whenever it closes, the PCM stream stops and lets the playout buffer drain. Discontinuous transmission in Opus additionally stops the stream during speech pauses, again letting the playout buffer drain.

[21] http://opus-codec.org/

[22] http://www.rowetel.com/?page_id=452

This is quite acceptable on voice, but sudden arbitrary timing jumps are unwelcome when copying CW. This hasn't been much of a problem on HF because the linear demodulator lacks a squelch and high background noise levels keeps Opus transmitting. The user may manually flush the playout buffer by hitting the 'r' key; having control of *when* a delay reset happens makes all the difference.

Timing jumps are very serious impairments in synchronous digital modulation, so again demodulators should always read the PCM multicast stream without lossy compression, activity detection or playout buffering. The *monitor* program is strictly for human listening.

AX.25 frames and the *packet* program

My first application for this package is to receive, decode and relay APRS fixes from the high altitude balloons we build and launch from Mount Carmel High School and the University of California, San Diego. This uses several more modules starting with the program *packet*, essentially a software receive-only implementation of the ancient KISS TNC.[23]

It reads a PCM stream from the *radio* program, demodulates 1200 bps AFSK and decodes HDLC frames. Valid frames are multicast for subsequent processing by other programs, including *aprs* and *aprsfeed*.

Aprs extracts APRS position reports from AX.25 frames and computes azimuth, elevation and range from the observing station. Eventually this will automatically steer directional microwave antennas.

The other program, *aprsfeed*, acts as a receive-only APRS i-gate[24] to the worldwide APRS network[25]. Unlike the *aprs* program, which must understand the mind-bogglingly gratuitous number of ways to encode positions in APRS reports[26], *aprsfeed* simply relays raw frames to an APRS network server without interpretation, performing only (some of) those operations required by the APRS servers such adding the reporting callsign to the address chain and dropping certain types of packets.

As with the PCM streams, these two programs also illustrate the utility of feeding the output of one module (the AX.25 frames from *packet*) simultaneously to several readers.

SDR hardware experience

Well-defined interfaces between components of a large project are at once vitally important and very difficult to get right. This project is as much about experimenting with those interfaces as creating components to do actual signal processing. Often the only way to find out if an interface will work is to try it, hopefully on a small enough scale that it can be changed without worrying too much about backward compatibility. My interfaces have already gone through several revisions and I won't pretend they're perfect. That's the whole idea: to build a proof of

[23] http://www.ka9q.net/papers/kiss.html

[24] http://www.aprs-is.net/IGating.aspx

[25] E.g., the popular http://aprs.fi website

[26] Don’t get me started about APRS protocols. Just don’t.

concept to experiment with so you can figure out what actually works.

I generate digital IF streams with standalone programs that talk directly to the front end hardware, usually by USB. So far I support the AMSAT UK Funcube Dongle Pro+ and the HackRF One with programs *funcube* and *hackrf*. Although I could probably support more hardware through SoapySDR [27], I often find such "shimware" packages less than satisfactory because their APIs often seem at once excessively general and complex yet incomplete by omitting ways to access certain hardware-specific features. (I do use the well-established *portaudio* library[28] to talk to digital audio devices, and I'm not sure even that was a great idea.) Besides, my program is arguably *itself* shimware, so why do it twice?

Initially I sent I/Q samples direct from the A/D converters, leaving it up to the consumer to set analog gains, remove DC offsets, and correct I/Q gain, phase imbalance and frequency errors. I've since moved those functions into *funcube* and *hackrf* to remove unnecessary hardware dependence from the programs that consume their digital IF streams. Because the HackRF One supports sample rates up to 20 MHz, I added optional decimation to *hackrf* to keep the Ethernet data rate reasonably low. I typically decimate a 12.288 MHz A/D sample rate 64:1 to 192 kHz, the same as the Funcube dongle. Other sample rates and decimation ratios can be selected if the Ethernet link (and the USB interface to the hardware) can handle them. Gigabit Ethernet should handle ~30 Ms/s sample rates assuming 16-bit complex samples, though this would probably require jumbo[29] Ethernet frames and careful attention to kernel buffering and CPU scheduling.

The decimator in *hackrf* can optionally shift the spectrum by one quarter of the A/D sample rate to move near-DC A/D artifacts well outside the output passband so the *radio* program need not avoid the DC region. In effect, this adds another stage of frequency downconversion and filtering.

A Raspberry Pi 3 handles the Funcube Pro dongle with ease because the sample rate is fixed at 192 kHz; even with DC offset, gain and phase correction it uses only 8.5% of one ARM CPU core.

The Pi's Achilles heel is relatively slow I/O; it cannot keep up with a HackRF sampling at 12.288 MHz so I'm currently using an old 1.66 GHz Intel Atom D510. Decimation filtering is with a cascade of 15-tap half-band filters implemented in Intel SSE and single precision floating point. 64:1 decimation takes 83% of one Atom core, and everything else (including offset/gain/phase correction) takes 50% of a second core.

Having started with the 16-bit Funcube dongle, I put off writing an AGC for some time. The HackRF One has only 8-bit A/Ds so I was finally forced to implement one. This AGC, being hardware specific, belongs in the *hackrf* program. Without it, *iqrecord* might record unusable data unless something else generates any necessary gain setting commands (*iqrecord* does not). A simple hysteresis scheme seems to work well. The analog gain settings are included in the metadata so *radio* can easily recover the (uncalibrated) absolute input signal level.

[27] https://github.com/pothosware/SoapySDR/wiki

[28] http://www.portaudio.com/

[29] https://en.wikipedia.org/wiki/Jumbo_frame

I would like to give *radio* signals calibrated to absolute levels at the antenna terminal but I found that SDR front end conversion gains are highly frequency dependent. This is especially so with the Funcube dongle because of its set of 11 preselection filters -- one of its major advantages in crowded areas. I would have to measure and construct a gain/frequency calibration table for each device.

Hardware contention resolution

Because digital IF streams are multicast, its easy to have multiple copies of *radio* listening to each stream, tuned to different frequencies and/or modes within the IF bandwidth. There's no explicit contention resolution, so *radio* sends a retune command to the SDR front end only when the user manually retunes it, and then only when necessary to bring the signal within the IF passband (i.e., between +/- one-half the sample rate). Because the first LO frequency is carried in the IF metadata, the *radio* programs will notice whenever the front end is retuned and will automatically adjust their own second (software) LOs to compensate, if possible. If the desired signal has moved outside the Nyquist bandwidth *radio* will patiently wait (unless the user manually retunes it). This avoids a possible “retuning war” between the *radio* instances.

An obvious way to lessen the contention problem is to use higher SDR sampling rates that can span, eg., an entire amateur band. Another might be to use a polyphase filter bank to create a pool of "virtual" front ends, each conveying some part of the input spectrum at a lower sample rate. The first approach is both simpler and conceptually more in keeping with the idea of multiple consumers sharing a common signal source but practical limits (like Ethernet link speeds) may come into play.

Problems with multicasting

No paper like this is complete without a candid discussion of problems and obstacles encountered, whether or not they were overcome.

IP multicasting admittedly has its share. Although the basic multicast protocols have been around for a long time, as explained earlier they have not been widely used beyond the simple, degenerate case of resource discovery on local area networks. These applications run at low data rates (rarely more than one packet per second) so they don't stress a network very much, and because they're confined to LAN segments they don't require any special routing mechanisms. So it wasn't a surprise to run into some problems when multicasting 6 megabit/s digital IF streams or in establishing multicast connectivity between LAN segments. There's always a price for living on the bleeding edge!

IP multicast routers are available as open source, but my experience with them has been frustrating. They are complex to set up, use Linux kernel features that are not extensively tested, and often require that you first set up a network tunnel (VPN) to provide logical adjacency between the two LANs you wish to join. This isn't too hard for (semi) permanent situations if you know what you're doing, but it has so far proven very frustrating to do on the fly, e.g., from a laptop through a public WiFi hotspot. I will probably resort to ordinary unicast traffic when I want to listen remotely to the audio from one of my SDRs.

Fortunately, explicit multicast routing is rarely necessary for most of my intended applications. The various elements all run on computers within my house, and most of those are in one room (my ham shack) where they are all connected to the same Ethernet switch. Multicasting is ideal in this case.

The common "dumb" (unmanaged) Ethernet switch handles multicasts in a very simple way: a copy is flooded to every switch port except the one on which it arrived.[30] They are handled just like a broadcast, which is just a special case of multicast. When the multicast rate is low, as it usually is in resource discovery, this creates no problems. Nor will higher speed multicasting cause any problems as long as the aggregate multicast data rate is below the spare capacity of the slowest port on the switch; every modern host Ethernet interface simply ignores multicast traffic in which it is not interested. But problems will obviously occur if, say, the multicast load is 20 Mb/s and one switch port is running at the old, slow 10 Mb/s rate.

Aside from doing without multicast entirely, this problem can be solved in several ways, not always satisfactory.

The simplest fix is to dedicate an isolated switch to high speed multicast traffic, avoiding any devices that cannot operate at full switch speed (usually 1 Gb/s). This may be a perfectly workable approach when the SDR modules are all running on closely located computers (e.g., in a ham shack) but it would be nice if it weren't necessary. Unfortunately, one especially slow Ethernet device is nearly ubiquitous on home networks: the WiFi base station.

802.11 WiFi uses an ever-widening range of transmission speeds to maximize the capacity of each radio link while minimizing interference to co-channel users. Link level acknowledgments are critical to finding the highest transmission speed that can be used to a given destination and in recovering packets lost to interference or sudden decreases in radio link quality.

This works fine for unicast packets, but most inexpensive WiFi base stations transmit Ethernet multicasts as *physical layer broadcasts* at a fixed low data rate without radio link acknowledgments. The multicast rate can sometimes be manually configured, but often it is fixed at just a few Mb/s. A single digital IF multicast stream at 6 Mb/s will obviously cause such a base station to roll over and die *even if it has no wireless clients who even want the stream!*

There are two highly effective fixes to this problem, but unfortunately they are rarely implemented in consumer-grade switches and access points.

The first fix is *multicast to unicast conversion.* Modern WiFi access points have such a wide range of transmission speeds that it

[30]This is inherent in Ethernet switching. *Any* packet to a destination address not in the switch's forwarding database (from being seen in the source field of some earlier packet) is automatically flooded out every port except the one on which it arrived. Multicast addresses never appear as source addresses, so they are always flooded.

is invariably more efficient to just send each recipient its own copy of each multicast packet using the usual unicast mechanisms at whatever speed the target can support.[31] Unfortunately, low end access points rarely implement this mechanism although support is increasing as the problem of multicast over WiFi become more widely known.[32] [33]

The second fix is *IGMP snooping.* A switch eavesdrops on the IGMP messages generated by host computers joining or leaving each multicast group so it can squelch multicast traffic to switch ports where nobody is listening. Placing a WiFi access point on an IGMP-snooping switch (and not having any of its clients subscribe to a high rate multicast group) is an effective fix. Some WiFi access points also do IGMP snooping.

IGMP snooping does have its drawbacks. [34] Although it seems to be universally implemented on “managed” or “smart” switches, less expensive models like my Netgear GS110TP don’t support IGMPv3, the latest version, forcing the network to fall back to an older, less capable version. IPv6 has its own form of IGMP called *Multicast Listener Discovery* (MLD) that my GS110TP doesn’t understand at all, so it can only filter IPv4 multicasts; IPv6 multicasts still flood to every port. This switch limitation seems so common that I don’t yet support IPv6 multicasting even though I would very much like to. (It also pays to read the fine print when buying a supposedly smart Ethernet switch or WiFi access point.)

A more basic problem is that IGMP snooping is a classic protocol layering violation: a layer 2 device (Ethernet switch) is looking at higher layer protocol (IPv4 or IPv6) information that it should treat as opaque data. But the performance benefits are so great as to warrant an exception.

[31] There’s a real irony here in that we’ve long seen radio as an inherently broadcast medium where, intuitively, it ought to be easy and efficient to have everyone receive a single transmission. Thanks to adaptive power control, modulation, coding and now MIMO beam forming, this “inherent” property of radio seems to be disappearing for good. The efficiency benefits of multicasting may be restricted to point-to-point (i.e., non-broadcast) wire and fiber channels! However, I do see a silver lining: surreptitious eavesdropping on a radio link won’t be as easy as it used to be.

[32] Many AT&T U-verse subscribers, including this author, first discovered WiFi’s vulnerability to high speed multicasts when they plugged their own consumer-grade access points into switch segments with Uverse set top boxes. A Uverse HDTV stream is about 6 Mb/s, the same as one of my digital IF streams, enough to saturate the access point until it is disconnected.

[33] https://tools.ietf.org/id/draft-ietf-mboned-ieee802-mcast-problems-01.html

[34] https://www.rfc-editor.org/info/rfc4541