SonATA Zero

A Prototype All-software SETI Detector

G. R. Harp, T. Kilsdonk, J. Jordan

Abstract

We describe the design and initial results from a prototype, all-software SETI detector which we call SonATAØ. SonATAØ simulates radio time series data and converts it to multicast Ethernet packets. These packets are received at a second server which "channelizes" the data, and returns 1024 channels or filaments to the network at different multicast IP addresses. A third server accepts one of the filaments, performs another level of channelization and displays a waterfall plot. This is a rudimentary approximation of the tasks that must be performed by the next generation SETI detector, called SonATA. SonATAØ serves as a basis for SonATA, and our development path is to incrementally improve SonATAØ until we reach full-up SonATA capabilities.

Introduction

The SETI Institute's next generation SETI detector (known as SonATA – SETI on the Allen Telescope Array), will eschew hardware signal processing in favor of an all-software design. This design, strongly recommended at the SonATA preliminary design review (Mar. 2005, Hat Creek, CA), requires that the Allen Telescope Array (ATA) emit phased-array radio data on a standard interface (IP packets over Ethernet). The SonATA system subscribes to this packetized data and carries out numerical processing in stages, as shown in Fig. 1.

In the figure, all the dark lines represent Ethernet connections. The first stage channelizer receives a time series of the radio signal at the ATA output rate (100 MS/s in full-up system). The channelizer may be implemented in one or more stages, with each stage connected to the previous via Ethernet. The channelizer cascade performs a numerical poly-phase filter on this data with a variable number of output "channels" (1024 nominal). Each channel contains a time sample stream at a lower data rate (e.g. 100 KS/s per channel). The channelizer acts as a programmable band pass filter on the input data, narrowing the 100 MHz input bandwidth to 100 KHz in each output stream, and generating a total of 1024 output streams.

After channelization, the 100 KS channels, or "filaments" are routed to a bank of something like 100 detectors. Each detector is capable of performing SETI search algorithms on a few dozen channelized filaments. The detectors search for narrow band continuous wave and pulse signals. The signal reports are analyzed by the control system software which has an automatic verification procedure for possible ET candidates. If a candidate passes all the tests, a human observer is notified.

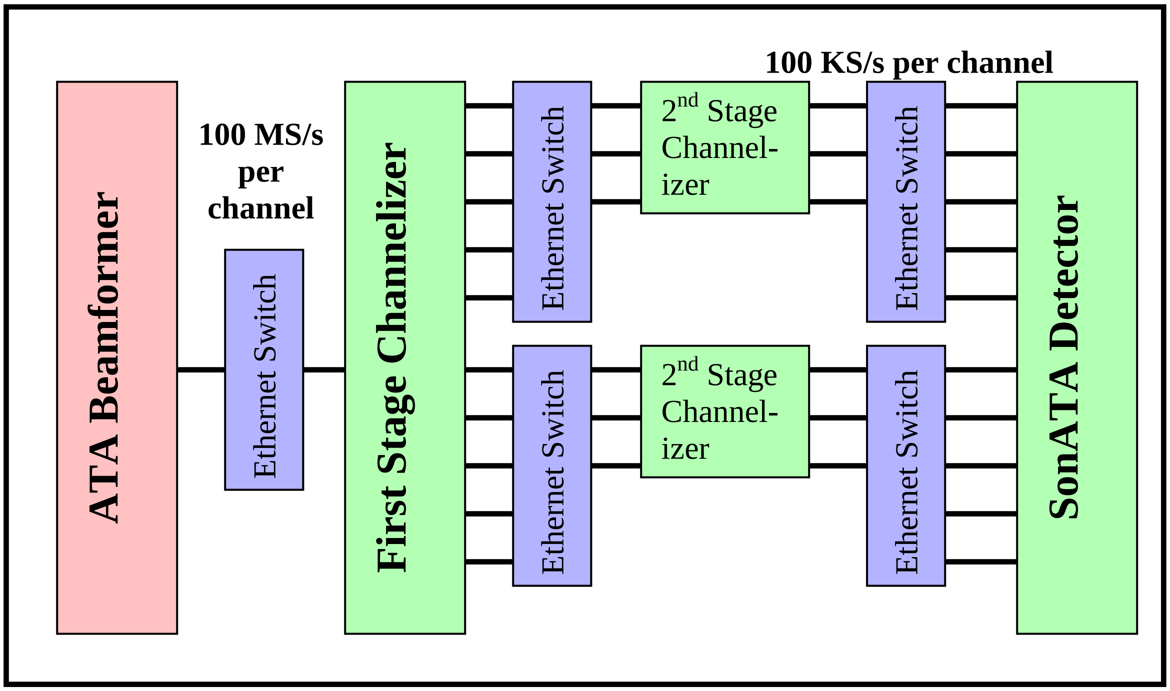

Figure 1: High-level block diagram of SonATA system. Each green box represents a different computer program, probably running on a separate computer server.

The current SETI search system (Prelude) already performs SETI detection algorithms in pure software on a consumer off-the-shelf (COTS) computer, but relies on proprietary hardware for channelization. The main development goals of SonATA are to migrate the channelizer into COTS hardware, and to rework the proprietary interfaces from telescope to channelizer, and channelizer to detector to be an Ethernet standard. Both of these tasks will require a serious development effort.

In order to begin this development, we have undertaken a prototype, or zeroth-level approximation, of the final SonATA system, using software and hardware tools readily available at the Institute. We call this prototype SonATAØ, and describe it here. SonATAØ is written entirely in Java, comprising 5 independent programs. SonATAØ achieves highest processing efficiency when run on three or more computers, but the programs can be run on a single computer at a reduced sample rate.

In many ways, SonATAØ forms the basis and framework for the final SonATA. Our development plan is to upgrade SonATAØ incrementally, with superior hardware and software, to gradually approach the full up SonATA. For example, the current Prelude detector will be modified to accept Ethernet packets and can plug in to the SonATAØ channelizer output right away. Alternatively, as soon as the ATA can produce Ethernet packets from its beamformer, these can be routed to the channelizer input, and SonATAØ can immediately display a waterfall.

By the time SonATA is complete, no single piece of SonATAØ may survive, and pieces may be upgraded multiple times. But starting with the present “release” of SonATAØ, we expect to maintain a continuous availability of SonATA functionality, all the way to the final system.

ATA Packet Definition

To begin the process we define the interface between ATA and SonATA. At the SonATA Workshop, March 11-12, Hat Creek, CA, John Dreher hosted a preliminary design review of the SonATA system. Our external reviewers, including Greg Papadopoulos (Sun Executive VP and CTO), John Wawrzyneck (UC Berkeley, Prof. EECS and BWRC), David Cheriton (Stanford, Prof. CS), Mike Lyle and others, strongly encouraged us to make the nominal ATA beamformer output a standard interface, namely IP over Ethernet. The reviewers suggested that we should consider multicast IP. Multicast has several benefits over singlecast IP, bidirectional IP, or some proprietary interface:

1. No handshaking is required, so traffic from ATA to SonATA is all one-way
2. No handshaking makes implementation simpler, especially for FPGA hardware
3. Multicast allows multiple users to "subscribe" to the same data stream without any increase in network traffic or extra hardware
4. Multicast allows the packet producers and packet receivers to come up and go down independently
5. Using an IP/Ethernet standard, all backend devices (e.g. pulsar machines, alternative SETI detectors) can accept this interface without requiring a lot of support from ATA staff
6. IP packets provide a straightforward path to making a "thin" data stream available on the internet, to allow up and coming SETI groups to develop their detectors

The full-up ATA beamformer generates about 3.2 Mb/s digital data (100 MS/s represented as 16b+16b complex samples), thus we will use 10 Gb Ethernet for this interface. As the beamformer evolves toward this goal, we expect to pass through several generations beginning with a single 1 Gb Ethernet link (compatible with SonATAØ). In the first generation, Girmay Girmay-Kaleta anticipates that the ATA beamformer digital signal will be passed through a 1-10 MHz band pass filter, and be emitted at tens to hundreds of Mb/s over the 1 Gb Ethernet interface. SonATAØ is designed to accept this sort of input, so we can use Girmay's prototype as soon as it becomes available.

To represent the ATA data, we choose a standard multicast UDP packet whose payload contains nominally, 1024 complex samples in 16b+16b format as follows:

ATA Packet Payload		
Name	**Bit Length**	**Bytes**
sequence	32b	4
firstchan	32b	4
numchan	32b	4
time	64b	6
cdata[]	(16b+16b) * 1024	4096
flags[]	8b * 1024	1024

Table 1: Definition of packets that are emitted from ATA beamformer.

The bit-lengths for various fields of this packet are chosen for programmer convenience. (Integer 32b is the natural size for numbers in Java.) The number of samples per packet was chosen after an empirical study of the transfer speed of packets over a 1 Gb network, summarized in Table. 2.

Complex Samples	Packet Length (Bytes)	Mbit / s	kPacket / s	Aggregate Data Rate (kSamples / s)
8	40	18	58	464
16	72	33	58	928
32	136	62	57	1824
64	264	118	56	3584
128	520	229	55	7040
256	1032	424	51	13056
356	1432	509	44	15664
512	2056	442	27	13824
1024	4104	553	17	17408
2048	8200	555	8.4	17203
4096	16392	570	4.4	18022
8192	32776	571	2.2	18022
16131	64532	563	1.1	17744
16384	65544	ERROR	ERROR	0

Table 2: Transmission rates of ATA multicast packets as a function of packet size.

The packets used in Table 2's study did not contain the flags array, but that doesn't change our overall conclusions. For small packets, the IP stack may pad the packets to a larger size or else there is a maximum packet rate which limits transmission rate. Packets greater than 64 kB in length are rejected by the network, and don't work at all. Packets with 1024 samples are large enough to benefit from maximum transmission rate and are not too large for convenient software processing. For this reason we choose 1024 samples in defining the prototype ATA packets. This interim choice will be revisited as we evolve to different computers and switch hardware.

In the future, we will use different network hardware (10 Gb Ethernet) and different software to manage ATA packets. In particular, we may take advantage of Layer 7 Ethernet switch technology. To see how this works, consider the contents of a multicast UDP packet (Figure 2). The Ethernet header and trailer are used only at the hardware level, and are transparent to the switch manager.

The IP (MAC) Header is used by low-cost Layer 2 switches you buy at Fry's. The IP Layer performs packet verification and takes appropriate action (either initiating resend or discarding the packet) if a corrupted packet is received. Thus high level applications can rely on packet integrity. Layer 2 switches read the MAC addresses (e.g. 00-13-CE-31-E1-4C) of sender and recipient from the IP header, and use this information to route packets from source to destination.

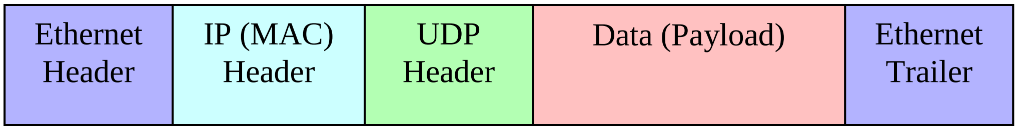

Figure 2: Description of UDP/IP/Ethernet packet contents.

A Layer 3 switch (router) is typically more expensive than a Layer 2 switch and is sometimes called managed switch. Layer 3 switches look into the UDP header of the packet, and discover the IP addresses (e.g. 128.32.24.48) of sender and recipient. Thus, if a computer is replaced with different hardware and assigned the same IP address, packets can still be routed to right destination in a transparent way. One wrinkle is that multicast packets have no specific MAC recipient at all. Instead, packets are sent to a generic IP address[*] so that many recipients can subscribe to the same packet. A Layer 3 switch is necessary in the SonATA system to route multicast packet to only specific computers that subscribe to particular data streams.

Layer 7 switches can additionally look inside the Data (payload) section of the packet. Thus packets can be routed to different destinations depending on their *contents*, which can be defined dynamically. In the full-up SonATA system, we may take advantage of this capability for load balancing across multiple detector servers. In the future we must augment the ATA packet described here with a standard UDP payload header that conforms to a specific real-time internet protocol such as RTPS[†], to support Layer 7 routing.

SonATAØ Packetizer

The first step is to simulate the output of the ATA beamformer. The real hardware beamformers are not expected until Spring, 2006, and the first generation Ethernet output from these beamformers is expected later. For this reason we simulate the beamformer output and store stock signal types in disk files. These files are transferred to a COTS computer where a Java program that transforms the signal into multicast packets on a 1 Gb Ethernet network.

Looking back at Table 2, when running at maximum capacity our packet server[‡] can emit approximately 500 Mb/s onto the network (lower columns of table). This is only half the speed supported by the network switch, and under these conditions all of the computer processing power is absorbed by the IP layer. We can be certain of this because in our tests we transmitted the same UDP packet, over and over, so the only processing done is to push this data onto the network.

[*] Multicast addresses are defined as those lying in the range 224.0.0.0 through 239.255.255.255. It is illegal to assign addresses in this range to a specific computer server.

[†] http://wiki.ethereal.com/Protocols/rtps

[‡] 2.7 GHz Intel Pentium IV CPU with 1 Gb PCI network adapter, plain vanilla server. All the numbers quoted for efficiency in this document are for one of 3 identical servers like this.

Last spring, Alan Patrick performed similar tests running both Java and C-code on identical machines, and obtained results similar to these and to one another. On this basis we conclude that there is no inherent difference between Java efficiency and C efficiency when talking to the network.

If the entire CPU is absorbed by network communication, there is no time left over to do processing. Hence our prototype Channelizer (next section) must run slower than the maximum rate in Table 2. In future generations, the packetizer will run on a more powerful machine, with multiple processors and 10 Gb Ethernet interface. We expect that as technology develops we can grow into a simulation of the full-up ATA beamformer (~4 Gb/s).

SonATAØ Channelizer

The role of the channelizer is to absorb a single time series of data (bandwidth B) from the network, break the time stream into N independent channels or "filaments" with bandwidth B/N, and serve these filaments back to the network. Because of the reduced bandwidth, if the input stream comes at R samples per second, each filament is down sampled to rate R/M where M ≈ N.[§] The break down process uses a poly phase filter bank (PFB), which is just a modified fast Fourier transform (FFT).

In SonATAØ the channelizer performs more processing than either the packetizer or detector, and this is the processing bottleneck. This balance may shift as the detector grows in complexity. The channelizer is software configurable in terms of the ATA packet size (1024), FFT length (1024), poly-phase filter order (9), and Backend packet size (512), where we indicate default values in parentheses. With these values on the present hardware (2.7 GHz Pentium IV, 1 Gb Ethernet), we reliably process 500 kS/s, or 500 packets per second from the packetizer. For discussion we assume the above values.

The channelizer does processing in three threads:

1. Receiver thread
    a. Receive multicast packets from network
    b. Check for missed packets, insert "zero" packet if one is missed
    c. Check for out of order packets, drop out of order packets
    d. Convert integer 16b+16b complex numbers to floating point
    e. Store floating point packets (length = 1024) in a Vector (FIFO)
    f. Check for Vector overflow (we're not keeping up)
    g. Repeat
2. Analysis thread (PFB)
    a. Retrieve packets from FIFO into length 9 array
    b. Multiply packets by FIR filter, length = 9 * 1024
    c. Sum resultant 9 packets into a single array, length = 1024
    d. FFT this array into complex frequency spectrum

[§] The downsampling factor M is usually smaller than the bandwidth factor N to make sure no signal is ever missed when it crosses from one filament to the next.

   e. Corner-turn frequency spectrum values into 1024 output packets, one value per packet, packets labeled by channel number
   f. After 512 iterations output packets are full, store finished packets in FIFO
   g. Repeat
3. Sender thread
   a. Retrieve output packets (length = 512) from FIFO
   b. Look at the channel number, calculate the destination IP address
   c. Emit packet to network
   d. Repeat

Now we discuss each thread’s activity in more detail.

Receiving Multicast Packets

Missed Packets: How safe is UDP communication? This depends very much on channelizer loading. When processing 500 packets per second, the channelizer misses about 1 packet in 10^4. As the channelizer speed is increased, packet misses become more frequent. Eventually the channelizer cannot keep up at around 1000 packets per second.

Our testing indicates that packet misses are mainly due to Java garbage collection. This happens because our simple program creates and destroys literally thousands of 5 kB packets per second. The Java garbage collector must run at a high priority because it is asynchronous with other processing, hence it sometimes preempts the processor when a multicast packet arrives.

One could argue that this is a drawback of Java, but that is slightly unfair because in other programming languages you must write your own garbage collection code whilst Java provides it for free. If the channelizer code were modified to reuse packets instead of needlessly creating and destroying them, then most of the packet misses would go away. Only testing can show if this reduces packet misses to an acceptable level (once acceptable is define), or if we must turn off garbage collection or choose a different programming language to meet our requirements. This is a task / design decision for SonATA.

Replacing missed packets: In the final SonATA, the choice of what to use for missed packets is another design question. For testing we found that replacing misses with all zeros was convenient, since it doesn’t substantially alter the test signal shape.

Out of order packets: After days and days of testing and literally billions of packets, we never observed a packet arriving out of order. Packet order is not guaranteed on a UDP network, but in our simple configuration it is no problem. If the channelizer encountered such a packet, we simply discard it in this implementation. Evidently this was a good choice, and substantially simplifies the receiver thread code.

Analysis Thread

Comparison of FFT and PFB

It is well known that the FFT is an approximation to a true Fourier transform because the input data is sampled over a finite time. Given a sinusoidal input to an FFT, and if the sinusoid period is an integer multiple of the time separation between samples, then the FFT of that signal will accumulate all of the power into one bin (Fig. 3, blue curve). But if the sinusoid period is not an integer multiple of the sample period, the FFT distributes power into every frequency bin (Fig. 3, purple symbols). In the purple curve, power “leaks” into many bins adjacent to the “real” frequency position of the signal.

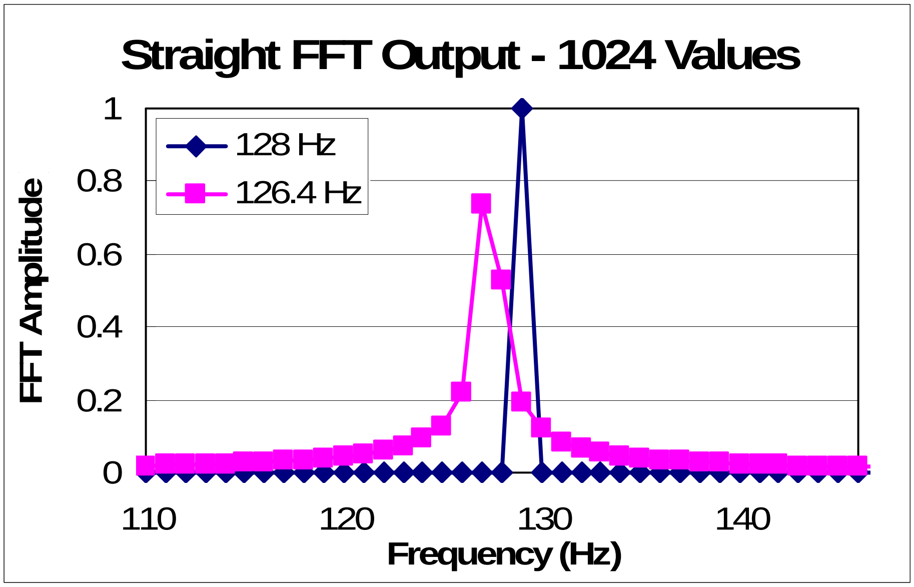

Figure 3: FFT power spectra from two 1024 length sample series (assuming 1024 samples per second). Blue: sinusoid period is integer multiple of sample period. Purple: sinusoid period is not integer multiple of sample period.

Comparatively, the poly phase filter bank (PFB) introduces a data filtering step prior to the FFT to reduce this power leakage. It starts with a longer time series, say 9x the FFT length, and then *simulates* a 9*1024 length FFT. Using a longer FFT confines the leakage power to a narrower region in frequency space. But the longer FFT gives us greater frequency resolution than we desire, so the PFB down samples the frequency spectrum -- binning it to the resolution of the original FFT. Binning is really a two-stage process: the finely spaced frequency data is first convolved with a “brick wall” function, and then every 9th sample is retained as the final output.

The above description is equivalent to a PFB, but a real PFB performs FFT and convolution in reverse order. Furthermore, the convolution is done the time domain instead of frequency domain using a bank of 1024 9-tap FIR filters. The 9*1024 coefficients for the FIR filter bank are generated from the inverse FFT of the brick wall function. Each FIR filter sums 9 time samples into one[**]; with 9*1024 input samples, we are left with 1024 samples which are fed to the FFT. This implementation is more efficient than the earlier description.

An example showing the performance of our PFB is shown in Figure 4. Here we use the same input sinusoid for both curves. The purple curve is the FFT, same as in Fig. 3. The green curve is the PFB output. The PFB causes almost all of the sinusoid power to appear in two bins, straddling the position of the input frequency, with low leakage.

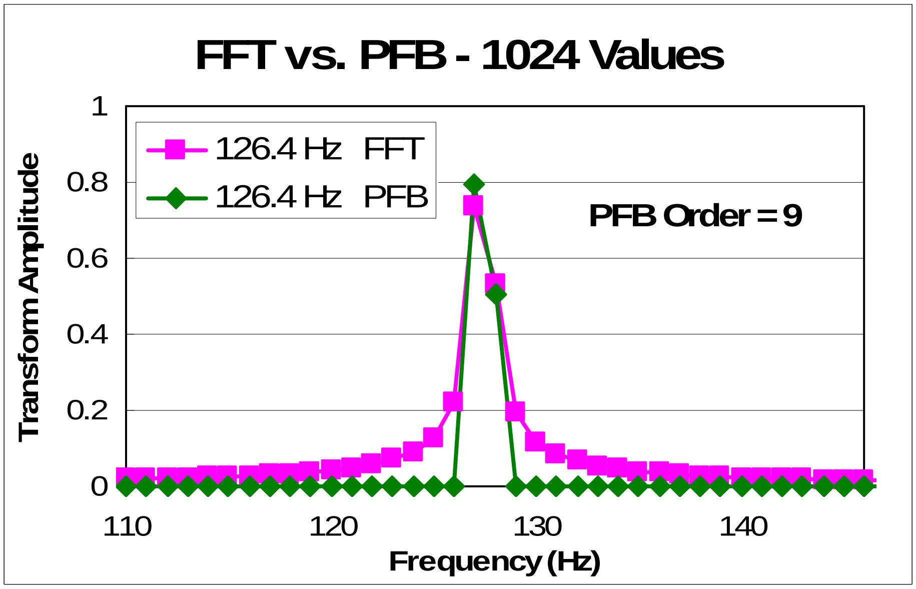

Figure 4: Comparison of a straight FFT (purple curve) and PFB (green curve) operating on the same sample stream. The PFB does a better job of localizing the signal power just to the two bins adjacent to the input signal frequency.

The FIR Filter Bank

The inverse Fourier transform of a perfect brick wall in frequency space has the form of a sinc function, sin(t)/t, and is infinite in length. A real PFB operates on a finite time series, so we must compromise the brick wall shape to some degree. A well-known method of managing finite-time effects is to multiply the idealized sinc function with a windowing function to smooth discontinuities at both ends of the truncated time series. We choose a Blackman window[††]. It is convenient to display the FIR coefficients as a single, interleaved time series, resulting in the curve in Fig. 5 (top).

[**] The 9 samples selected for a given 9-tap FIR filter are chosen to be 1024 samples apart in the input time stream. That is, the indexes of the samples for the first FIR filter are (0, 1024, 2048, …, 8192), for the second FIR filter are (1, 1025, 2049, …, 8193), etc.

[††] Blackman window = 0.42 + 0.5 * cos(2.0 * x) + 0.08 * cos(4.0 * x), (-π <= x <= π), and the x coordinate represents a scaled time or sample number. For sample 0, x = -π. For sample 9216, x = π.

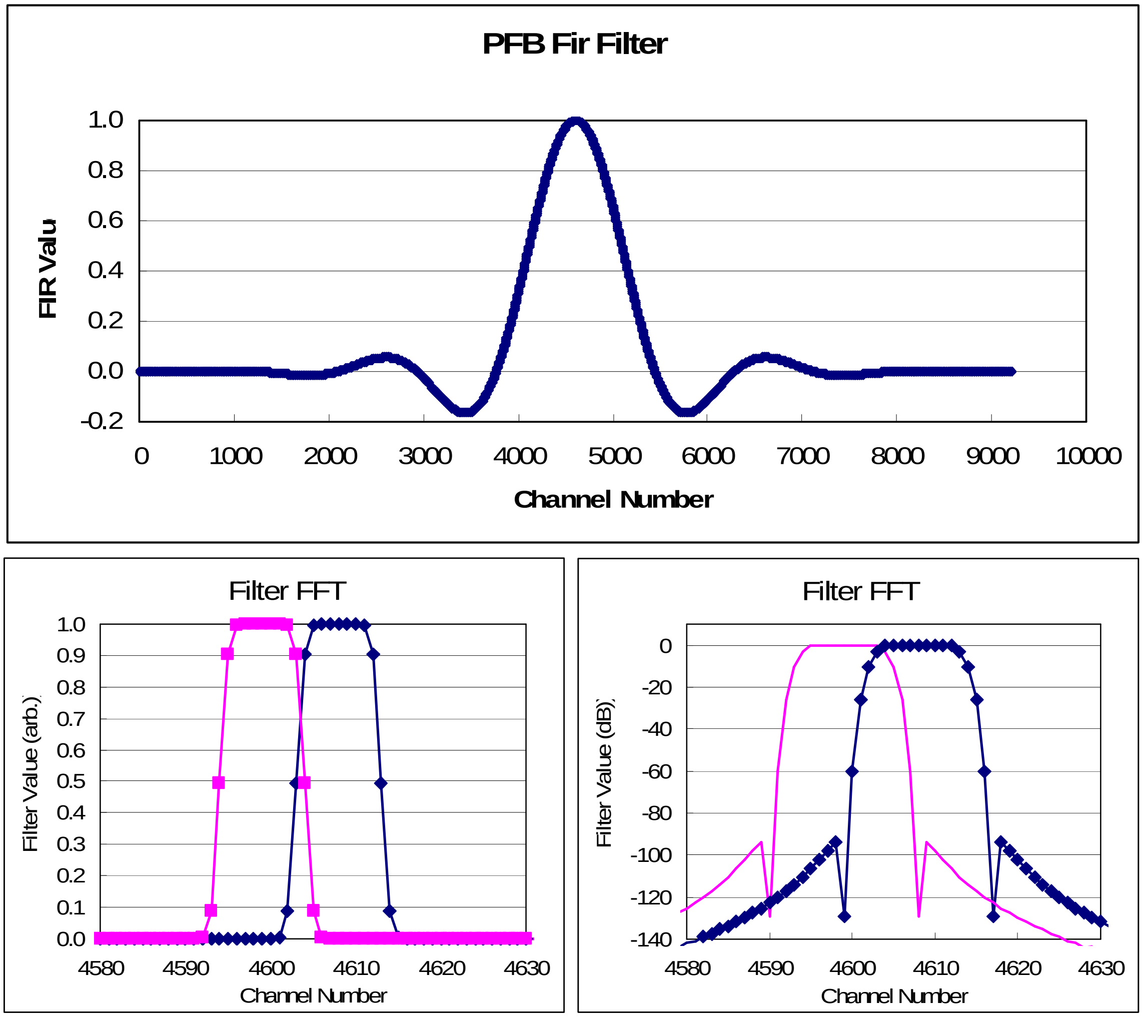

Figure 5: Top: Graph of the FIR coefficients used before the FFT in the PFB. Bottom: Two views of the FFT power spectrum of the top curve, in the vicinity of the channels where there is substantial power.

To visualize the brick wall shape we FFT the curve in Fig. 5 (top) and show linear (Fig. 5 bottom left) and logarithmic (Fig. 5 bottom right) views of its power spectrum. To understand these curves, suppose that we have performed a 9*1024 FFT on a section of the time stream. The blue curves show the convolution function operating on the finely-spaced frequency spectrum. After convolution we keep every 9th point, so we display a second copy of this curve shifted by 9 points (purple). The frequencies under the purple curve contribute to one output filament. The frequencies under the blue curve contribute to an adjacent filament.

Notice that output filaments are not perfectly independent; for example, a signal appearing midway between two filaments will appear with 70% power level in both of them. Blackman windowing gives a very flat top to the brick wall and good isolation between adjacent channels (greater than 80 dB image rejection at half-channel point). Of course, all of these qualities can be tuned by varying the PFB order or choosing a different windowing function.

One nuance is that the actual FIR coefficients used are equal to the ones plotted above but alternating in sign. This sign alternation in front of the FFT causes the “natural” frequency order of the FFT (0, 1, 2, …, Ny-1, -Ny, -Ny+1, -Ny+1, … -1) to be flipped into “human” order (-Ny, -Ny+1, …, 0, 1, …, Ny-1). Here we have labeled frequencies by their index, and Ny = Nyquist frequency.

FFTW Library

As part of the ATA software, we previously developed a Java Native Interface to the FFTW library, 2.1.5. This circa 2003 version of FFTW does not support the latest microprocessors but is still quite efficient. It is a floating point FFT library that supports both single and double precision (we use single). The latest version (FFTW 3.0.1) has a substantially different interface, so the Java wrappers would need a few days of rework to bring them up to speed.

Sender Thread

This task is quite straightforward except for the large number of separate filaments emitted from the channelizer. Ideally, we send each filament to a different multicast IP address, so that the detector can subscribe to exactly the data of interest. But we discovered two limitations associated with Linux and/or the current Java implementation. The first limit is in the sockets layer. A single socket may join (can be associated with) no more than 20 different multicast addresses or groups. This is really no problem; we worked around it by having the program open multiple sockets with 16 addresses each.

The second limitation is that under Linux 2.2 there was once a kernel limit of 1024 open files at a time. Although this limit is surpassed in Linux 2.4, it is still mirrored in the Linux implementation of Java. We discovered that no more than 1024 multicast groups may be joined in any Java program. Since one group is the source of ATA packets, we worked around this problem by sending 2 output filaments to each of 512 multicast addresses. Since outgoing packets contain their channel number, the detector program must examine the channel number discard packets from uninteresting channels.

The 1024 separate address problem can be addressed in many ways. We recently learned to send multicast packets, it is not really necessary to join the multicast group. Therefore this point may be no problem as long as we only send to 1024 addresses, but we haven’t tested this. Another is that on a Sun operating system, this limit may not exist. But most importantly, it is unlikely that the channelizer will operate with 1024 output filaments, at least for a year or two (see below). As we have shown, 512 output filaments work fine.

Making It Faster

The SonATAØ channelizer is not especially efficient, but we used the best tools available that we could integrate easily. We spent a few hours profiling it and tuning its performance. As mentioned above, most of the processing time is spent in the analysis thread. In Table 3, we display the results of some crude profiling measurements that take into account *only* the analysis loop:

FIR and Copy	FFT	Corner Turn	Allocation
45%	35%	14%	6%

The most process-intensive task is the “FIR and Copy” task, which takes the 9 input packets, does a 9k floating point multiplications with the FIR filter, and then sums the channels from the 9 packets into a single one for input to the FFT. This should not be surprising considering that the FIR and each of the packet arrays are allocated in different regions of memory. A more careful implementation of this code section might make a substantial improvement. Note that the complexity of this task depends on PFB order.

However, we point out that until we reach the final goal of full-up Sonata[‡‡], we will usually not use a 1024 point transform. Given a 10 MHz time series from a down sampled beamformer, we are more likely to do 128 or 64 point transforms in the channelizer so that the output filaments are still at the desired ~100 kS/s rate. To see how this might change the equation, we ran a few tests using 64 point transforms instead of 1024. In that case, the “FIR and Copy” task dropped to only 1/3 of the FFT task, which itself should increase by a factor of 2. This supports our suggestion that memory management is at the heart of FIR and Copy slow down.

As for the FFT, as mentioned above we expect logarithmic speed up with decreasing FFT length. Additionally, John Dreher predicts a factor of 2-3 speed up if we switch to FFTW 3.0.1, since that library uses vector operations.

In an earlier section, we discussed allocation and garbage collection. With our crude tools we did not measure the time spent in garbage collection, but did measure the time to allocate packets transmitted to the send thread. This allocation (and associated garbage collection) could be avoided by reusing packets. We can expect a performance increase of something like 10% with this, relatively simple, coding change.

Finally, a multi-processor system like the Sun V40Z can offload the packet receiving and sending tasks from the processor doing the analysis. This can also offer a substantial improvement in processor efficiency.

To conclude this section, we find that with a relatively small effort and new hardware it should be possible to speed up the present channelizer by a factor of a few. Inserting highly-optimized processing routines developed by Rick Stauduhar or others will improve things even more. We predict that it will be challenging, but feasible, to achieve 10 MS/s processing on a one (or two staged) server(s) in 2006.

SonATAØ Detector

The SonATAØ Detector program leverages a sophisticated package of FFT and display routines, written in Java, and originally developed for NSS and Prelude. It consists of two programs, the first of which extracts a single filament of time-series data from the output

[‡‡] Even in the full-up SonATA, channelization might be performed in e.g. 2 stages, in which case the FFT length would be 32.

of the channelizer and writes it to a file. A second program displays this file as a waterfall plot (frequency on horizontal axis, versus time on vertical axis, with power as intensity) using a slightly modified version of the existing NSS/Prelude waterfall display program. A snapshot of the waterfall display is shown in Fig. 6.

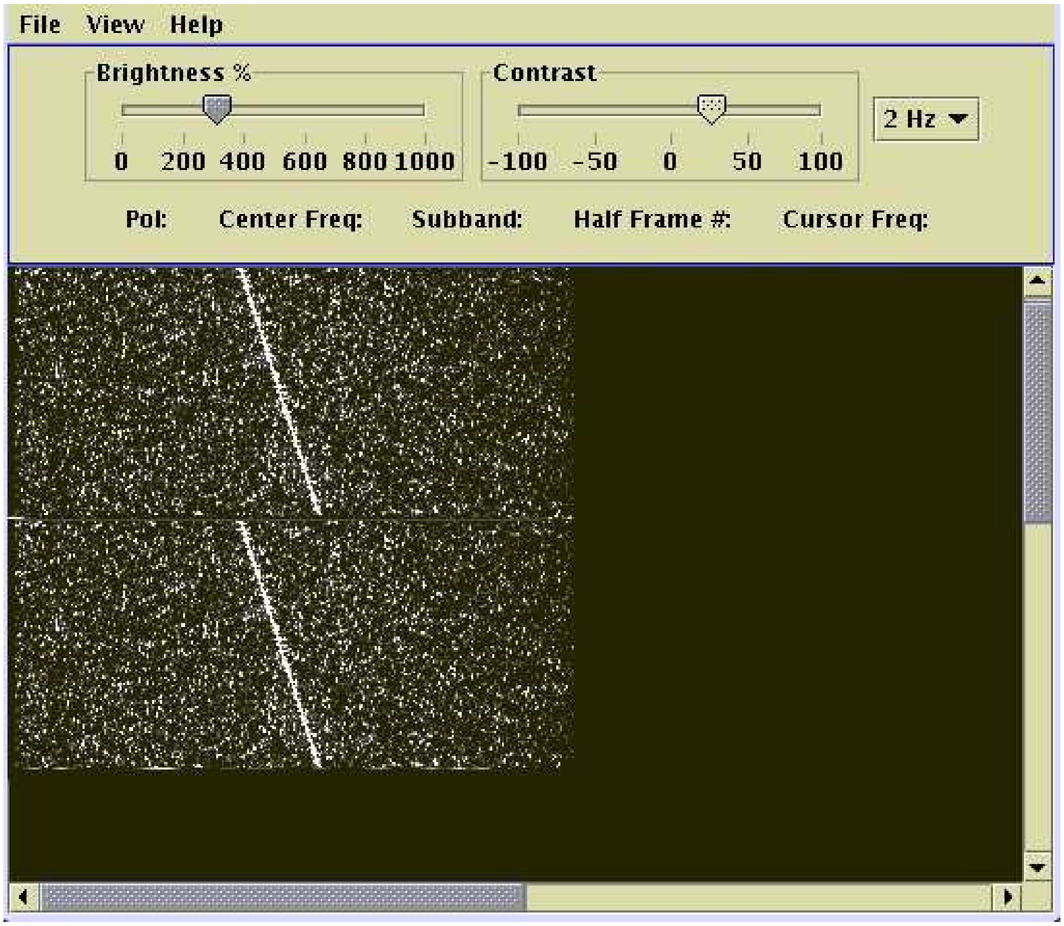

Figure 6: A screen shot of the waterfall display, which is the principle SonATAØ detector. The white noise, or "static" in the display is simulated noise in the radio receiver. The diagonal white line arises from a drifting test tone in the data, similar to a possible ET signal.

Contemplation of waterfalls like Fig. 6 lead one to ask, "What kind of time stream would be necessary to cause a picture to appear in the waterfall display?" This effect can be obtained with the assistance of sleep deprivation on long observing runs, but we have a simulator, too.

Starting with Fig. 7 top as input, we perform an inverse-FFT on a suitably padded[§§] copy of each raster line in the image. This becomes the input time stream for SonATAØ. The lines are fed to the packetizer one after the other, channelized, and finally detected in the waterfall plot (Fig. 7 bottom). This waterfall suggests how Google may one day advertise an encyclopedia galactica to extraterrestrial observers.

[§§] Each raster line is padded to 524288 length with samples of Gaussian white noise.

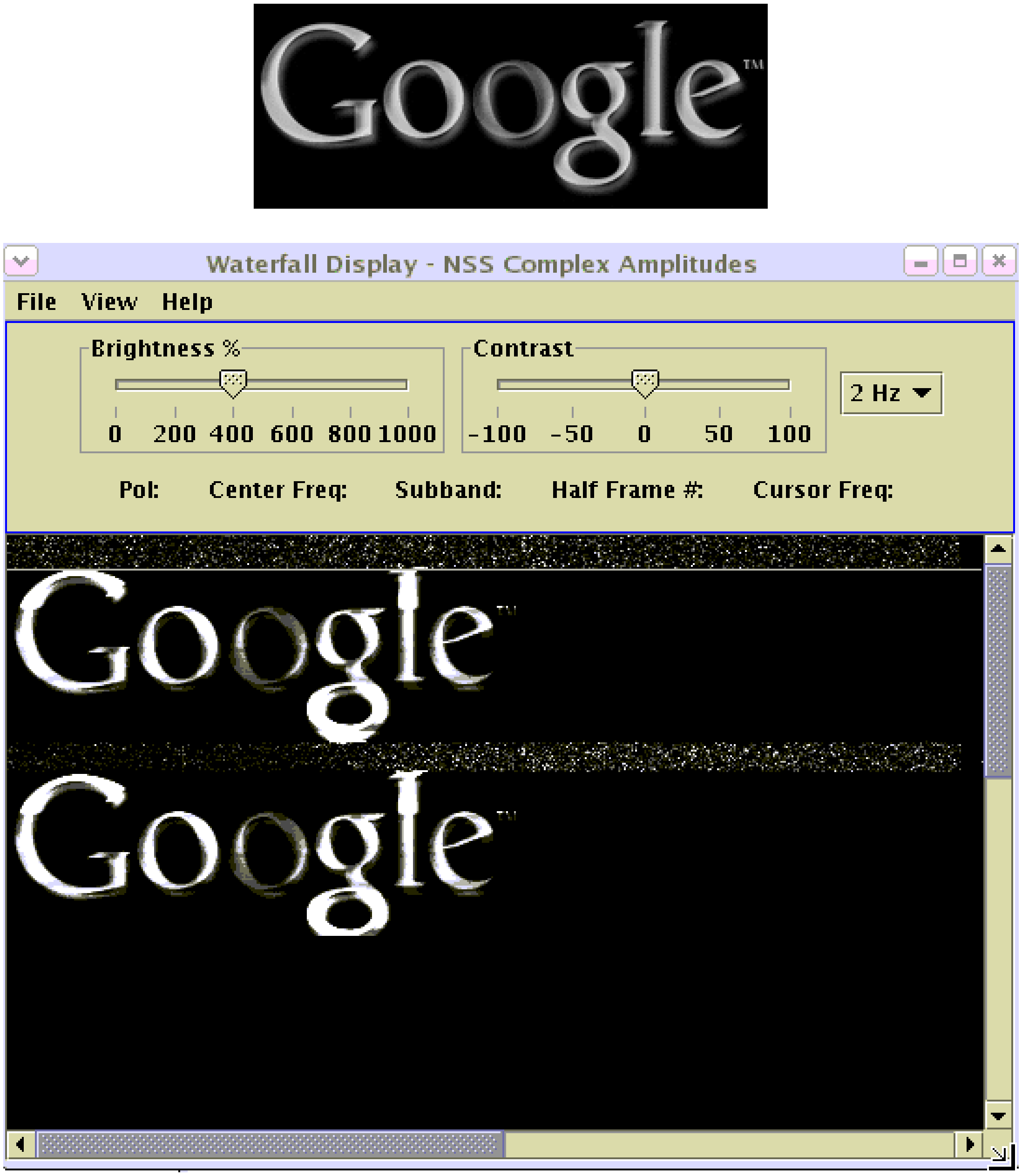

Figure 7: Top: this image is used to generate a time stream for the packetizer. The time stream bears little resemblance the image because it contains an inverse FFT of the raster lines. Bottom: After a poly phase filer in the channelizer and a second FFT, image reappears in the waterfall.

The packetizer sends the same time stream over and over, so Fig. 7's waterfall shows multiple copies of the input image. There are a few features to be explained. Firstly, the waterfall is intentionally configured with a high contrast to highlight weak signals. It also performs an auto-scaling operation which gives rise to banding in the image. Although the image is padded with white noise, auto scaling causes the noise to disappear at times when the image is very bright.

Lessons Learned

Socket programming is especially easy in Java, and remarkably portable. The multicast sending/receiving programs were developed and tested on a Windows machine. They were ported without change to a Sun box and packets were sent from Windows to Sun. Before long, we copied the same programs to a Linux platform and ran them there. This is an area where Java shines; as a prototyping language. The SonATA project expects to take delivery of some very fast servers in the new future, thanks to a generous donation from Sun. As soon as they are set up, we expect zero development time to migrate SonATAØ to the new platform.

Head to head comparisons between C++ and Java sockets show that they achieve identical performance, both in speed and in CPU loading.[***] This is surely because Java calls down to native, kernel routines for UDP operations.

Because we are calling into the FFTW library (highly tuned C), the processing speed for this step is not significantly different from the same program written in another language. As mentioned above, it might make sense to migrate from FFTW 2 to FFTW 3 to obtain a performance enhancement. On the other hand, highly tuned code from Rick Stauduhar may soon obsolete FFTW in this application.

Although the packetizer emits multicast packets and the channelizer receives them, in the present configuration the communication is point to point. The ATA beamformer output is currently planned as a multicast interface, but if singlecast UDP might be another option if implementation requires it.

Conclusion

SonATAØ is a prototype of the next-generation detection system for the SETI Institute. SonATAØ demonstrates how the synchronous time stream of the Allen Telescope Array can be turned into asynchronous Ethernet packets, and delivered to a network with many back end observers. The SonATAØ channelizer captures a wide bandwidth sample stream, executes a poly phase filter, and returns 1024 narrow-bandwidth filaments to the network. SonATAØ detectors recover these filaments, perform more processing and display the filament contents in a waterfall plot.

SonATAØ serves a dual purpose. It is a proof of principle for several new technologies that we will use in the full-up SonATA. But more importantly, it is the seed from which the final SonATA will grow. In a very real sense, SonATA is already here. It only gets better and better with time.

[***] After testing by Alan Patrick, reported in personal email to one of the authors (GRH) dated 7/5/2005.