VoxNet: An Interactive, Rapidly-Deployable Acoustic Monitoring Platform

Michael Allen
Cogent Computing ARC
Coventry University

Lewis Girod
Ryan Newton
Samuel Madden
MIT/CSAIL

Daniel T. Blumstein
Deborah Estrin
CENS
UCLA

Abstract

Distributed acoustic sensing underlies an increasingly important class of sensor network applications, from habitat monitoring and bioacoustic census to security applications and virtual fences. VoxNet is a complete hardware and software platform for distributed acoustic monitoring applications that focuses on three key goals: (1) rapid deployment in realistic environments; (2) a high level programming language that abstracts the user from platform and network details and compiles into a high performance distributed application; and (3) an interactive usage model based on run-time installable programs, with the ability to run the same high level program seamlessly over live or stored data. The VoxNet hardware is self-contained and weather-resistant, and supports a four-channel microphone array with automated time synchronization, localization, and network coordination. Using VoxNet, an investigator can visualize phenomena in real-time, develop and tune online analysis, and record raw data for off-line analysis and archival. This paper describes both the hardware and software elements of the platform, as well as the architecture required to support distributed programs running over a heterogeneous network. We characterize the performance of the platform, using both microbenchmarks that evaluate specific aspects of the platform and a real application running in the field.

1. Introduction

Acoustic sensing is a key component in a broad range of sensor network applications, including gunshot localization [23], weapon classification [28], acoustic intrusion detection [5], biological acoustic studies [17, 30, 3], person tracking [25], speaker localization [8], and smart conference rooms [29]. Existing work has described specific algorithms, and in some cases custom hardware. This paper is about a flexible hardware and software solution that can potentially target any of these applications, evaluated based on a specific bio-acoustic application.

The specific application we consider relates to the problem of automated census of *in-situ* animal populations using bio-acoustic signals such as animal calls [6, 27, 9, 16]. This application requires sophisticated localization, signal enhancement, and classification algorithms to process the acoustic time-series data, as well as statistical algorithms to process the resulting animal call event traces. In some cases automated census may require combining acoustic detections with other sensor inputs such as imagers triggered based on timely results from the acoustic data processing pipeline. While algorithms have been developed for census and population measurement for many specific species [17, 6, 27, 9, 16], for some species and environments this is quite challenging, and general solutions remain elusive.

Because acoustic monitoring applications present many challenges that are not readily addressed by existing platforms [20, 19], previous work in acoustic applications has typically required significant investment in platform development [28, 17, 5]. Several key challenges complicate the development and deployment of acoustic applications:

- Acoustic data is generated and processed at high rates, placing a heavy computational burden on the sensor platform. Even with large memories and fast 32-bit processors, the processing pipeline must be highly optimized to perform well.
- Developing on-line processing algorithms generally requires pilot studies to collect sample data. These pilot studies are often similar to the eventual deployment in terms of the logistical efforts and hardware functionality required.
- While some acoustic applications require on-line processing, others require retrieval of complete traces, or some balance between the two, perhaps dynamically configurable at run-time. Both on-line processing and

storage contribute to system load and impact platform requirements.

The design of our platform, called VoxNet, stems from considering these observations in the context of bioacoustics applications, with a particular focus on animal tracking and census problems, but with the potential to be applied in other domains thanks to a flexible, deployable hardware platform and a high-performance distributable programming interface.

Software flexibility is required to support multiple concurrent applications, and in particular to allow reconfiguration and tuning of applications running in the field. For example, concurrently with continuously running an animal call localization application, a researcher might want to archive raw data or test out a new detection algorithm without disturbing the running system. To achieve this we implemented an interface for VoxNet that allows new programs to be installed in a running system without interrupting existing applications. Distributed VoxNet applications are written as a single logical program, abstracting the programmer from the details of the network and particular hardware platforms. These programs can operate over a combination of live and static data, residing in a distributed system of sensors and backend servers. This model enables users to tune and further develop applications during pilot deployments, and enables the system to be used as an interactive measurement tool while it is deployed. This is important for many short-term scientific deployments, because it allows a scientist to immediately explore newly observed phenomena. Installing program updates at run time is also useful in other contexts such as security monitoring.

Usually, greater run-time programmability incurs a cost in performance. To address this, VoxNet builds on prior work designing the WaveScript [10] language and compiler, which we extended to support the sensor interfaces and network functionality of VoxNet. VoxNet is the first embedded target for the WaveScript compiler, and developing the VoxNet backend motivated many new features and optimizations. Implementing our animal localization application using the WaveScript programming language and optimizing compiler results in a 30% reduction in processor load and 12% in memory usage, compared with the hand-coded C implementation used in a previous version of the application (see Section 5).

Thus, the main contributions of this work are:

1. To develop a platform capable of rapid deployment in realistic environments for bioacoustic applications;

2. To provide a high level programming interface that abstracts the user from platform and network details and compiles into a high performance distributed application; and

3. To define an interactive usage model based on run-time installable programs, with the ability to run the same high level program seamlessly over live or stored data.

2. Related work

Our related work is concerned with both platforms for network tasking, and processing frameworks for high-frequency domains (acoustic, seismic).

2.1. Platforms and architectures

Tenet [14] advocates a tiered networking approach to in-network processing, where less capable platforms have clearly defined roles and are tasked by more powerful platforms. The reasoning behind this type of architecture is that the complex aspects of a computation should stay on more powerful platforms, because it is less costly and error prone than push the computation to less capable platforms.

Tenet's goals differ from ours. In the reference implementation, Tenet nodes are Mica2/TelosB class devices, and micro servers are Stargate class processors. In contrast, our lowest-tier sensor nodes must be much more capable than Mica or TelosB motes to deal with high-frequency data, and we do not enforce a boundary between processing tiers. In fact, one of our aims is to make this boundary more transparent, so it can feasibly adapt to changes in application and environment.

VanGo [15] is a system which is designed to capture high frequency phenomena using devices which are constrained not only by processing capability, but also network communication bandwidth. A focus of VanGo is therefore to support data reduction after suitable application-specific characterization.

VanGo forms a program as a linear chain of filters, designed to reduce data from its original form into events of interest. These filters can be enabled or disabled, and there are a library of different filters, such as FIR, event detection and classification. Again, our work focuses on a different class of devices than VanGo, and the programming model provided is, accordingly, much more general. Whereas VanGo is limited to a linear chain of filters, VoxNet allows an arbitrary dataflow graph and operators are not limited to filter semantics.

Mate/ASVM [24] provides a framework for application specific virtual machines (ASVMs), to allow developers to created customized runtimes which are application dependent. Network reprogramming is a key focus of virtual machines such as Mate. In our target domains, as well, phenomena are often not well-characterized, and application requirements may vary over the lifetime of the network. However, we choose native code compilation for our programs in order to achieve the level of performance required for intensive signal processing. Because VoxNet has a relatively

fast wireless network, uploading new binaries on demand is tractable.

The **Acoustic ENSBox** [13] is an ARM-based embedded platform for rapid development and deployment of distributed acoustic sensing applications. The VoxNet node hardware is based on the ENSBox hardware design, albeit greatly improved. Software is developed for the ENSBox using the Emstar [11] framework, and provides services such as time synchronization [7] and self-localization. However, multi-hop IP networking support is not provided on the ENSBox, and so multi-hop communication is provided through flooding interfaces, which does not scale well. This also means TCP connections cannot span multiple hops. Several deployments have used the Acoustic ENSBox and its related software support to record time synchronized acoustic data for offline bioacoustic analysis, or run on-line event detectors. However, due to the heavy load placed on the embedded CPU, and the latency incurred in writing to flash memory, it is not possible to run both of these applications in tandem. It was also not possible to perform other signal processing operations whilst the online event detector was running. Due to the optimizations of WaveScript, VoxNet can enable more functionality whilst incurring less CPU and memory overhead.

Figure 1. The packaged VoxNet node, shown in deployment at the Rocky Mountain Biological Laboratory, August 2007.

2.2. Processing tools and frameworks

There are many signal processing tools to carry out acoustic research. However, the selection of systems that support real-time processing, or embedded and distributed processing, is much more limited.

General purpose frameworks. Matlab [2] provides a general purpose mathematical environment and programming language and includes many specialized "toolboxes", such as for signal processing. Labview [1] is a similar application. Labview allows data acquisition and processing through a dataflow language. Applications are built using a graphical interface, enabling non-programmers to write acquisition and processing software, and visualize the results. However, neither Matlab nor Labview are well suited to implementation in a distributed system of sensors, because they are too inefficient to be implemented on embedded hardware and they do not have good support for implementing distributed applications. The VoxNet platform represents our initial steps towards creating a distributed, bioacoustic system that is easy to use and productive for the scientist as Matlab.

2.3. Applications

Acoustic-based census is a viable option because many mammals and birds produce loud alarm calls, territorial calls, and songs that are species-specific, population-specific, and often individually identifiable [26]. As such, these vocalizations can be used to identify the species present in an area, as well as in some cases to count individuals. Acoustic monitoring for census has been shown to be important for cane-toad monitoring [6], elephants [27], birds [16] and whales [9]. Although the variety of species and habitats makes these problems difficult to solve in a general way, we believe that our platform and programming environment can be used as a substrate upon which all of these applications could be built.

3. Motivating Application

To make the motivation for VoxNet more concrete, we consider a specific use case with which we are familiar. In previous work, we developed the Acoustic ENSBox platform [13] and developed an application for localizing marmots, a medium-sized rodent native to the western United States [3]. In this deployment effort we worked closely with field biologists who are interested in studying rodent alarm call patterns to better understand their behavior. Working with real users and a concrete application enabled us to refine our objectives, both in terms of the capabilities of the system and the desired user experience.

In our usage scenario, a team of biologists want to detect marmot alarm calls and determine their location at the time of the call, relative to known burrow locations. Because alarm calls are a response to predator activity, they are typically quite rare and unattended operation is desirable. Although for some applications simple recording and off-line

processing would be sufficient, in this case it is important that the system process the data on-line to produce timely results. When biologists *are* present in the field, timely results from the system enable them to record additional data about the current conditions, e.g. what caused the alarm, and which animal raised it. Future versions of the system might automate this enhanced data collection in the event of a detection.

Since biologists are the "users" in our scenario, it is crucial that the system be easy for them to deploy, start up, and configure. The VoxNet node hardware is a compact, self-contained and weather-resistant package, as shown in Figure 1. To support non-expert users the VoxNet software is designed to be be easy to configure: a self-localization system [13] supplies fine-grained estimates of location and array orientation, a self-organizing ad-hoc network provides IP routing back to the gateway, and an easy-to-use web-based interface assists in troubleshooting. The goal of a VoxNet deployment is to work "out of the box" (with only minimal trouble shooting), after node placement and power up. The user can then install an alarm call localization application from the control console and visualize the results.

The algorithm for localization, described in [3], first processes a stream of audio data through a "fast path" detector to identify possible alarm calls, estimates bearing to the caller from multiple points (using the Approximated Maximum Likelihood or AML algorithm [3]), and finally fuses those estimates together to estimate the caller's location. To implement this on the VoxNet platform, this algorithm is expressed as a WaveScript program, a logical dataflow graph of stream operators connected by streams. The distribution of operators to nodes is made explicit by program annotations. Once constructed, the program is compiled and pushed out to the network to run; results are streamed back to data storage components and operational statistics stream back to the console. The data flowing on a stream can be visualized by connecting a visualizer to a viewable stream endpoint.

From our experiences with field scientists, we have found that even in instances where detection and localization can be done on-line, the scientists also want to record the raw data for future processing. While this desire may diminish as they gain confidence in data reduction algorithms, it will always be important in the testing phases of any new algorithm, as well as for interactive use to replay and re-analyze recent data. To address these concerns we have implemented "spill to disk" functions by adding new WaveScript operators that can save a stream to persistent storage for future retrieval. In a VoxNet deployment, network limitations mean that raw data can only be saved in a node's local storage (and as such can still be accessed by a distributed WaveScript program). After the deployment, stored raw data can be dumped to a large server for archival purposes; the same programs that run in a deployment can be run against the archived raw data.

4. The VoxNet Platform

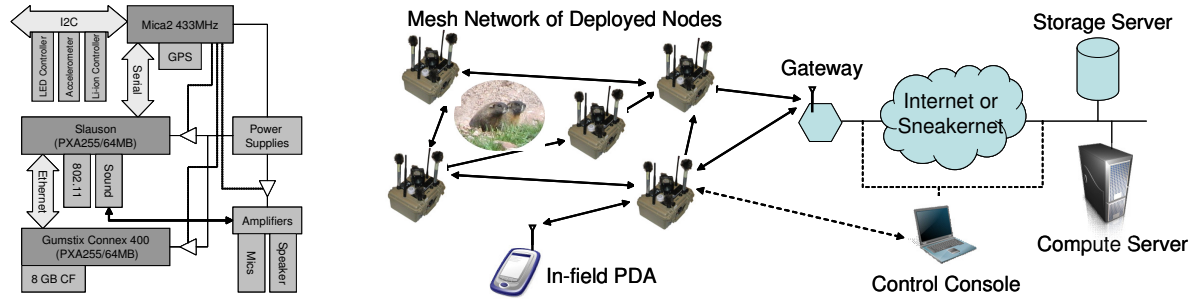
The VoxNet platform consists of an embedded node platform and a software platform that supports distributed applications, as described in Section 3. Figure 2(b) is a diagram of the VoxNet system architecture, a framework in which programs can be written, compiled, and disseminated, and the results can be archived and visualized. In this architecture, a user runs a program by submitting it through a user interface at the control console. The control console discovers the capabilities of the nodes in the network, and assigns portions of the Wavescript program to different nodes based on program annotations. It then compiles and optimizes each distributed component of the program for the appropriate platforms, and disseminates the compiled program components to the network of embedded VoxNet nodes and backend service machines. Results and diagnostic data are returned to the control console for display and visualization; a PDA may also be used to visualize data while in the field. Streams of results and offloaded raw sensor data can be archived to a storage server and later processed off-line, using the same user interface.

The next two subsections describe the hardware and software components of VoxNet in greater detail.

4.1. Hardware

The basic node hardware, shown in Figure 1 and represented by the block diagram in Figure 2(a), is a revision of the Acoustic ENSBox prototype [13]. The VoxNet node shares the same main processor board as the original Acoustic ENSBox, based on the Sensoria Slauson board, a 400 MHz PXA 255 processor with 64MB memory, 32MB onboard flash and two PCMCIA slots containing a 4 channel sound card and an 802.11 wireless card. VoxNet nodes currently use an ad-hoc mesh network using 200 mW output 802.11b cards and 5.5 dBi antennas. These cards have a maximum range of approximately 200m in line-of-sight conditions. VoxNet nodes currently consume 7.5 W continuous when running, and have an 8-hour lifetime from an internal 5400mAh Li-ion battery.

Storage Co-processor. Hardware limitations of the main processor board precludes the use of large storage devices. To address this, VoxNet contains an auxiliary processor, the Gumstix Connex 400 with the NetCF I/O board, that is connected to the Slauson by wired Ethernet. The Gumstix hosts a large Compact Flash card that can archive data streams from the Slauson. Since the Gumstix board runs the same VoxNet software as the Slauson, a WaveScript program can easily be configured to offload portions of the computation to the Gumstix.



(a) Hardware Architecture

(b) System Architecture

Figure 2. VoxNet hardware and system architecture. WaveScript programs invoked at the command console are compiled and disseminated to the network of sensors; returning results are visualized and archived to a storage server. In cases where the deployment is disconnected from the backend services, results and raw data is archived to node-local storage and transferred when the nodes are brought on-line.

Supervisory Co-processor. Duty cycling is an important feature, given the 8-hour lifetime of VoxNet nodes. VoxNet includes a Mica2 [19] that is always on, linked to the Slauson board via a serial line. The Mica2 manages the indicator LEDs, external attention button, GPS, accelerometer and temperature sensors, and controls power to the other components of the system, including the Gumstix, Slauson, the audio output power amplifier and audio input preamplifiers. Software on the Mica2 allows the Slauson to power the system down for a specified interval and awaken from a button press. Currently the Mica2 radio is not used, but future software versions might implement over-the-air wakeup via the CC1000 radio.

Packaging. The VoxNet node is a self-contained unit installed in an 1150 Pelican case, with the microphone array integrated into the lid of the case. All external connectors are weather-resistant and sealed. The microphone modules detach and pack inside the box for shipping. The total weight of a VoxNet node is 2.3 kg.

VoxNet nodes have been used in several deployments over the last year in which weather conditions were unfavorable, including a rain-forest in southern Mexico and a mountain meadow in Colorado. The microphones are protected by a waterproof latex cap and an artificial fur wind screen. During both deployments the nodes survived repeated exposure to precipitation.

4.2. Software

The VoxNet software is made up of three components, which we describe separately: Wavescope, which handles compiling, optimizing, and running an application on a distributed collection of nodes; control and visualization tools, which provide a collection of features important for tracking resource availability, troubleshooting and usability in the field and in off-line computations; and VoxNet platform drivers and services that adapt the general-purpose components listed above to the specifics of the VoxNet platform.

4.2.1. Wavescope and WaveScript

The Wavescope [10, 12] project is an ongoing effort to develop a stream processing system for high-rate sensor networks. This project includes multiple stream-processing execution engines for various platforms, united through their common use of a custom programming language, *WaveScript*. The WaveScript language is described in more detail in previous work. For our purposes here we will briefly outline its features and provide a single code example—namely, the main body of our marmot detection and localization program shown in Figure 3.

WaveScript is a stream-processing language. Like most stream-processing languages, WaveScript structures a program as a set of communicating stream operators (also called kernels or filters). In WaveScript, the user writes a "script" that composes together stream operators to form a graph. Thus the variables in the code of Figure 3, are bound to stream *values*, and the functions build stream *dataflow graphs*. The script to compose a stream graph can include arbitrary code.

A single script includes the full application code that is distributed over multiple tiers of the network. In the example code, the `netSendToRoot` function inserts a placeholder in the stream graph that marks where the graph is split by the compiler into node and server programs. In future work we may address automatic decomposition of a stream graph across network tiers. Note, that in this example, the node sends *both* a stream of raw detections and processed detections back through the network. Detections

```
(ch1,ch2,ch3,ch4) = ENSBoxAudio(44100)

// Perform event detection in frequency domain
freq = fft( hanning( rewindow(ch1, 32)))
scores = marmotScore(freq);
events = temporalDetector(scores);

// Use events to select audio segments
detections = sync(events, [ch1,ch2,ch3,ch4])

// Now we create a stream of booleans indicating
// whether the queue is too full for local AML.
queuefull =
     stream_map(fun (percent) { percent > 80 },
                AudioQueueStatus())

// Then we use that boolean stream to route
// detections into one of two output streams.
(iffull,ifempty) = switch(queuefull, detections)

// If full, AML on server, otherwise node
aml1 = AML( netSendToRoot(iffull))
aml2 = netSendToRoot( AML(detections))

// Once on the server, regardless of how we
// received AML results we process them.
amls = merge(aml1,aml2)
clusters = temporalCluster(amls)

// All AML results in a cluster are merged
// to form a likelihood map.
map = fuseAMLs(clusters)

// We route these likelihood maps back to the
// user ("BASE") for real-time visualization.
BASE ← map
```

Figure 3. A script for composing a graph of stream operators for the full marmot application (node- and server-side).

are processed using the AML algorithm [3]. Which path is chosen depends on whether the node has the free time to execute the required AML computation locally, which in turn is a function of the state of the data acquisition queue. We will return to this topic below.

A unique aspect of WaveScript is that it also allows one to write custom stream operators, inline, in *the same language* that is used for the script itself. A trivial example of this appears in the code when we use `stream_map` to apply a user defined function over the values in a stream. Other operators allow stateful processing of streams, and can be used to write custom, high performance signal processing operators, when needed.

Although it has always been intended for sensor networks, the prior published results of Wavescope have been aimed at high performance stream processing using desktop or server-class hardware. In porting Wavescope to VoxNet, we, for the first time, began to address some of the unique considerations encountered on a resource-constrained sensor node platform. As our sensor nodes do not possess processor cores, began by choosing the single-threaded ML-based WaveScript backend. (Other backends generate C code but include some overhead for multiprocessing support.) This WaveScript backend uses an aggressive, whole-program optimizing Standard ML compiler (MLton) to generate machine code. In addition to the good baseline performance of this compiler, the WaveScript implementation ensures that extremely intensive kernels of computation (for example, Fourier transforms) are out-sourced to the appropriate C or Fortran implementations (such as FFTW or LINPACK).

Further, the WaveScript backend we chose provides a simple model for executing dataflow graphs that is appropriate to our hardware and our target applications. It performs a depth-first traversal of the dataflow graph; emitting data from an operator is a simple function call into the code for the downstream operator. This is the right model for bioacoustic applications, as the data passed on streams is relatively course grained (e.g. windows of signal data), and because intra-node concurrency is not an issue.

To combine WaveScope with VoxNet, we leveraged WaveScript's foreign function interface (FFI) to integrate our audio hardware and networking layer. In particular, we wrote C-code that acquires data and pushes it to the compiled WaveScript code, where it is viewed as a stream. Our driver for the audio data-source use a separate thread to poll the audio hardware and queue data for the compute thread (WaveScript). We also decided to publish the status of the queue itself (percent full) as a streaming input to the WaveScript program. This enables the WaveScript program to make adaptive decisions about, for instance, whether or not to process data locally or ship it off (see Figure 3).

Taken together, Wavescope and VoxNet show that it is possible to program a signal processing system efficiently using a high-level programming model. Our system work in integrating VoxNet and WaveScope need not be repeated for subsequent users or deployments; they need only utilize the WaveScript programming interface. This interface exposes high-level stream and signal processing operators, and in addition, all WaveScript user code takes advantage of automatic memory management and high-level ML-like language features—while maintaining excellent performance.

4.2.2. Distribution, Control and Visualization

VoxNet implements network streams and a range of control and visualization tools to support the dissemination of applications and the display and management of results. Since prior work on Wavescope had focused on the language and single-node engine performance [10, 12], the VoxNet platform motivated significant new development, including a network stream subscription protocol, a resource discovery service, a control console interface, application dissemination, and visualization tools.

Network Stream Subscription. To support distributed programming in VoxNet we implemented Wavescope network streams, a network stream abstraction based on a publish-subscribe model and designed to operate over multihop

wireless networks. A Wavescope network stream closely mirrors the semantics of local Wavescope streams, in that they are reliable and allow fan-out to multiple readers. To support reliability we built this mechanism atop TCP streams, and added an automatic reconnect and application-layer acknowledgment to guarantee reliability in the face of disconnection. We also implemented timers that kill and restart the TCP connection in the event that it stalls due to repeated packet losses stemming from poor signal quality or temporary routing problems. The network stream abstractions are used for all types of communication in the VoxNet system, including visualization, log messages, control messages and pushing new compiled programs.

Although 100% reliable streams are required to preserve the semantics of local Wavescope streams, they are not always a practical solution. Consequently we have implemented extensions to this model that allow for certain types of "lossy" streams. A Wavescope network stream buffers all data until acked at the application level by each connected client. However, the buffer is limited to a fixed upper bound (defaulting to 512KB) and further data queued will cause stream elements to be dropped from the head of the queue.

Another important lossy semantics is called "always request latest". In this model, all new clients begin receiving only new data, and no buffered data is kept. In this semantics, the TCP stream will guarantee that all data is received during a given connection, but some data may be dropped if the connection is forced to restart. The raw data archiver uses this type of stream, because in the event of a disruption, the raw data will quickly overrun any in-memory buffer, and because the wired connection to the Gumstix is very reliable.

While the current implementation uses end-to-end TCP sessions, in other application contexts and at larger scales this may no longer be adequate. Further work is required to experiment with other communication mechanisms including "split-TCP" and DTN approaches. We also expect that other semantic models will arise as we further develop VoxNet applications.

Discovery Service and Control Console. The control console is a centralized point of contact for the entire VoxNet network, that discovers nodes, installs applications and tracks resource usage, error logs, and profiling statistics. It serves as a mediator between users who want to install a program and the VoxNet distributed system, and hosts all appropriate compiler tools and scripts. The discovery service hosted on the control console maintains a list of active VoxNet nodes and backend server machines, and tracks their capabilities and resource availability. When VoxNet nodes or servers start up, they connect to the control console at a well-known address and register with the network.

When a user submits an application for propagation to the VoxNet system, the control console compiles it for the appropriate architectures and pushes the compiled components out to nodes currently in its discovery list. The current implementation pushes each binary separately to each node using end-to-end TCP connections. This is acceptable for small networks, but as these systems scale, other mechanisms such as viral propagation or methods similar to Deluge [18] will be needed.

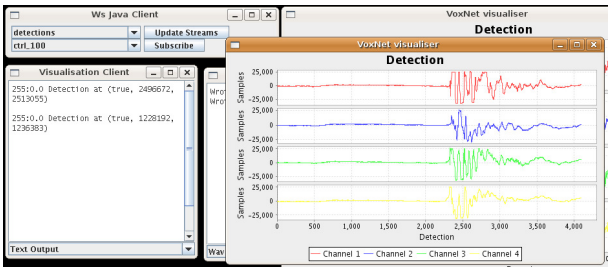


Figure 4. Example screenshot of time-series data in our visualizer.

Control Tools. To control our VoxNet deployment we developed the Wavescope shell, a text command-line interface to the control console, similar to the UNIX shell. This shell allows a user to submit new programs and see log messages and results streaming back from the network. Individual commands can be pushed to the nodes, either individually or as a broadcast, to start and stop the engine or to control other aspects of their behavior. To help visualize the network, the shell provides a "scoreboard" showing the current status of each node registered via the discovery protocol. In the current implementation the Wavescope shell is integrated with the control console software, but future versions will separate those functions.

Stream Visualization. Visualization of intermediate data and results is a critical component of making VoxNet usable. Visualization clients are Java applications that can run on any network-connected client machine, such as a laptop or a PDA. The Java visualizer connects to a Wavescope published stream and represents the data visually in real time. We are using the JFreeChart graphing package to build these visualization tools.

Currently we have developed several visualizers for different types of stream data, including the simple time-series visualizer shown in Figure 4 and a polar plot visualizer. The current implementation requires that the visualizer component exactly match the stream data type, meaning that in general an adaptor must be implemented in Wavescript to convert source data to the appropriate type stream; we are investigating developing a visualizer that can read an arbitrary marshaled wire protocol. Since the control console maintains a list of all currently available streams from each node in the network, interested clients can request this list

and thus browse for streams of interest.

Spill to Disk. Support for persistent storage is crucial to many uses of the VoxNet system, whether because of limited RAM buffer space, or to support offline or delayed analysis of raw data. The spill to disk component saves properly time-stamped raw data streams to the large flash in the VoxNet node or to disks in the case of backend storage servers. In the VoxNet node, the Slauson adds an extra operator to its running dataflow program that publishes the raw data as a network stream. A subscription client on the Gumstix co-processor reads that data over the network and marshals it to files on the flash, properly annotated with global timestamps. In our experiments we have found that VoxNet can continuously archive 4 channels of audio at 44.1 KHz while concurrently running other applications (see Figure 5 in Section 5 for details).

4.2.3. VoxNet Platform Drivers and Builtins

In addition to the reusable components described above, there are many important platform elements that are specific to VoxNet. Many of these are hardware drivers, diagnostic software, and glue components that are too detailed to be described here. We mention below two of the more complex and important features specific to the VoxNet platform.

Time Synchronization and Self-localization. VoxNet inherits time synchronization and self-localizing system developed for the Acoustic ENSBox [13], and adds additional glue to integrate these features into the Wavescope engine. Reference Broadcast Synchronization [7] is combined with a protocol to propagate global time from a node synced to GPS [22]. Timestamps in VoxNet systems are converted to global time before being transmitted over the network or saved to persistent storage.

IP Routing. VoxNet implements IP routing from each node back to a gateway node using a user-space implementation of DSR [21], that dynamically installs entries into the kernel routing table. DSR can establish routes between any two nodes on-demand, however in our field experiments we only needed multihop routes along the tree between nodes and the gateway. VoxNet uses 802.11 radios in ad-hoc mode, and enables a prism2 chipset specific *Psuedo-IBSS* mode to eliminate network partitions due to different parts of the network converging on different ad-hoc cell IDs. To eliminate the complications in configuring client devices to use *Pseudo-IBSS*, the gateway forwards between the *Pseudo-IBSS* network and a normal wired or wireless lan.

5. Performance and Evaluation

In this section, we investigate our platform's performance in-situ with respect to our motivating application, and also perform some microbenchmark evaluation. Our microbenchmarks are intended to validate of our choice of using Wavescript and the Wavescript compiler opposed to a hand-coded C implementation.

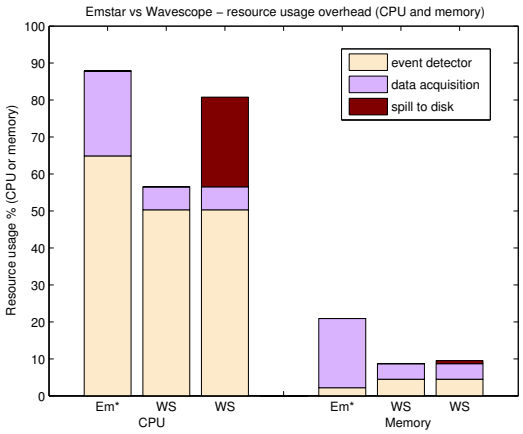


Figure 5. Resource usage comparison of the identical event detectors, implemented in Emstar and Wavescope.

VoxNet's approach allows us to blur the processing boundary between sink and node. Our in-field experimentation highlights the trade-offs that can be made with respect to in-network processing.

5.1. Microbenchmarks

Memory consumption and CPU usage. We compared the resource footprint (CPU and memory usage) of the event detector application described in section 3 with our previous hand-coded, Emstar implementation [3]. Figure 5 shows a breakdown of the relative components involved in the event detection implementation—the detector itself and the data acquisition, in terms of memory and CPU usage. These figures are the mean values of one minute's analysis of CPU and memory usage using the Linux command *top* (20 data points).

Figure 5 compares the footprint of the Wavescope application to that of the Emstar version. The graph shows that the total overhead of the Wavescope version is over 30% less in terms of CPU (87.9% vs 56.5%) and over 12% less memory (20.9% vs 8.7%).

Spill to disk. Because the Wavescope version uses fewer CPU and memory resources, additional components can run concurrently with the detection application. To demonstrate this, we verified that "spill to disk", a commonly used component that archives a copy of all input data to flash, could run concurrently with our detection application. We ran the event detector program and spill to disk simultaneously for 15 minutes, monitoring the CPU and memory usage on the

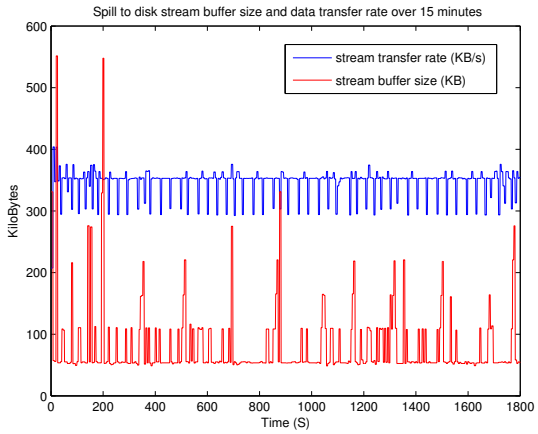


Figure 6. Data transmission rates between Slauson and Gumstix, as well as buffered data queue size, over a 15 minute period.

Slauson. We also logged the data transfer rates between the Slauson and Gumstix boards and the buffered queue sizes for the incoming data stream at one second intervals. The results in Figure 6 show that the mean data transfer rate from Slauson to Gumstix over the stream was 346.0 KB/s, which is in line with the required rate of 344.5 KB/s. This also accounts for the timestamping overhead of the raw data segments (8 bytes per segment). The third and sixth bars in Figure 5 show that the overall resource overhead of running both spill to disk and an event detector on the node is 80.7% CPU and 9.5% memory (taken from 20 consecutive *top* measurements). Given that the spill to disk feature was impossible to implement in Emstar due to resource constraints, we see that VoxNet's improvements in efficiency enable a corresponding improvement in functionality.

On node processing comparison. Providing the capability for on-node data processing is an important part of VoxNet. To further test our platform's processing capability, we measured the time taken to compute a direction of arrival estimate using the Approximated Maximum Likelihood (AML) algorithm [3]. In our motivating application, this is an intensive processing operation.

We compared the performance of a C implementation of AML (previously used in an Emstar-only system) to a corresponding Wavescope implementation. We invoked both implementations of the AML computation on the output stream of a Wavescope event detection program and timed how long they took to complete (measuring the start and finish of the AML function call in both cases). For both implementations, 50 detections were triggered to make 50 AML calls. Table 1 shows the min/mean/median/max comparison of processing times. For comparison, figures are

	Min	Mean	Median	Max
C (node)	2.4430	2.5134	2.4931	2.7283
Wavescope (node)	2.1606	2.4112	2.4095	2.5946
C (x86)	0.0644	0.0906	0.0716	0.2648
Wavescope (x86)	0.0798	0.1151	0.0833	0.5349

Table 1. WS vs C AML processing times

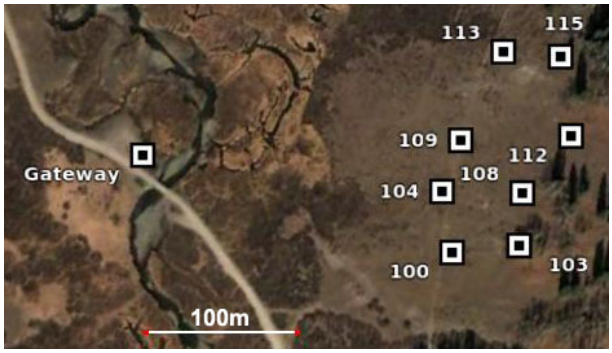


Figure 7. A map of the deployment area. The gateway was 200m away from the nodes.

shown for the same AML computation running on an x86 laptop-based version of the VoxNet node [4], with 256MB RAM and a P4 2GHz processor.

We see comparable performance between C and Wavecript generated C in both ENSBox and x86. We expect the performance of Wavescript generated code to be at best as efficient as hand coded C, so this result is encouraging. Both implementations used the same O3 optimization flag, as to ensure a fair comparison.

5.2. In-situ Application Tests

In August 2007 we tested the VoxNet platform in a deployment at the Rocky Mountain Biological Laboratory (RMBL) in Gothic, CO. We deployed the event detection and localization application previously mentioned in section 3. Over the course of several days, eight nodes were deployed during the day and taken down at night. We maintained a consistent geographic topology in each case, spanning the 140m by 70m area (2.4 acres) shown in Figure 7. We positioned a gateway node 200m away from the nearest node in the system (node 104). From this position, the control server accessed the nodes via a wired connection on the gateway. The control server ran a Wavescope shell to control the nodes, as well as the AML and postion estimation parts of the marmot localization application. It also logged all messages generated within the network and sent over control streams.

During our deployment time, we performed tests with two different sized antennae—standard and extended.

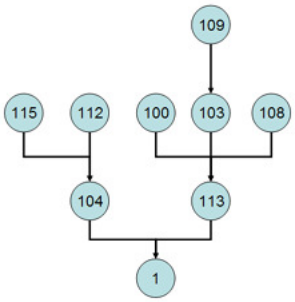


Figure 8. Routing tree for multi-hop testing.

Throughout all of these tests, we found that two of the nodes were consistently better-connected to the gateway than others (nodes 113 and 104). We speculate that this was a function of favorable geographical position relative to the gateway in terms of line of sight and elevation.

Our deployment was very much an exploratory exercise, allowing us to understand more about the behavior of our system in-situ, and help inform the kind of latencies we could expect for data transfer, detection processing and position estimation—all important aspects to help the scientist's observations at a higher level.

Goodput measurement. On two separate days, we ran goodput tests, in which a command requesting all nodes to simultaneously send a user-specified payload of data was issued at the control server. Each node measured the time when it began to send, and inserted a timestamp the first 8 bytes of the payload. A second timestamp was recorded at the sink when it completed reception of the payload. We used these time differences as an indicator of both the goodput (the transfer rate of application level data) and the latency of each transfer. Our tests took place in both a single hop network (with seven nodes) and a multi-hop network (with eight nodes).

We chose data payload sizes which were indicative of the raw detection and processed AML data traveling over the network (32 KB and 800 bytes respectively). This was designed to roughly mimic the behaviour of the network upon all nodes simultaneously triggering a detection and transmitting, albeit with slightly more controlled transmission times. In our one-hop test, we used a larger antenna on the gateway to achieve this one-hop network; the standard size antenna could not reach all nodes in a single hop. Getting a single hop network was useful as it allowed us to test the network over a larger area but without any problems arising from multi-hop communication. The graph in Figure 9 shows the min, mean and max latency of the time taken by each node to transmit 32 KB. These figures were taken from 16 individual data requests.

We expected to see similar results for each node, which

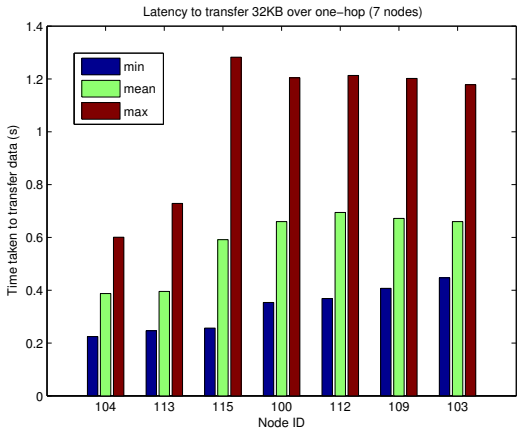


Figure 9. Transfer latency for 32 KB, one-hop

is generally the case, although it is clear that nodes 104 and 113 have smaller overall latencies than other nodes, consistent with our observation of their better overall connectivity to the gateway.

Our second goodput test was carried out in an eight node multi-hop network with up to three hops. The routing tree is shown in Figure 8. Statistics were gathered over 15 distinct tests, over a 287 second period (around 4 3/4 minutes). The data sent by each node per request was 32000 bytes. We expected to see goodput performance degrade with hops, and this is especially obvious in Figure 10. In fact, it is so pronounced that we see clear differences in latency between the different branches on the routing tree. The two nodes that route through node 104 (112 and 115) have lower latencies than the three that route through node 113 (108, 109 and 100), and node 109 (which is three hops away from the sink) incurs the most latency.

General operating performance. To examine regular application performance, we ran our marmot event detection application over a 2 hour period (7194 seconds), during which time the nodes in the network were triggered 683 times (in total) by marmot vocalization. Only 5 out of 683 detections were dropped (a 99.3% success rate). Although we expected 100% data reliability, we observed drops due to the overflow of our 512K network buffers during times of network congestion and high detection rates.

During one deployment day, a rain storm occurred whilst the nodes were running. Raindrops hitting the microphones and node cases caused the event detectors to constantly trigger. We expected that the network would become heavily congested, and that nodes would drop data as their transmit queues increased. Over the course of just 436 seconds (7 1/4 minutes), 2894 false detections were triggered, for a rate of around 6/second network-wide.

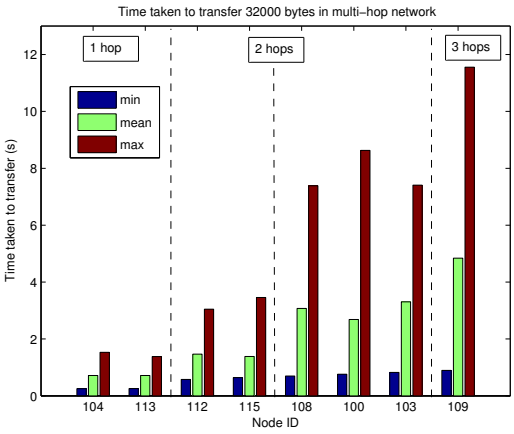


Figure 10. Transfer latency for 32 KB over multiple hops.

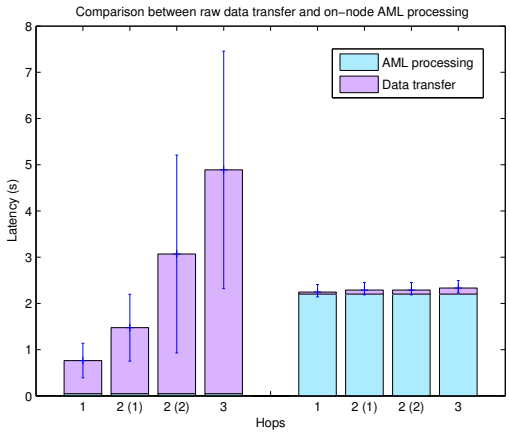


Figure 11. The latency trade-off between transmitting 32 KB and 800 bytes.

Each node dropped a significant amount of data during the raining period, only successfully sending around 10% of data generated to the sink (10.44MB). Despite the drops, our system continued to run, showing that it can deal with overloading in a graceful manner.

On node vs sink processing. In Section 5.1 we demonstrated that VoxNet nodes are indeed capable of processing the AML part of the call localization algorithm locally, although at a much slower rate relative to a typical laptop: the VoxNet nodes process an AML in 2.4112 seconds on average, versus around 0.0833 seconds on an x86 laptop. However, the result of processing a detection using AML is only 800 bytes in size, compared to a 32 KB raw detection. The difference in speed is therefore traded off by a reduction in network cost, as well as by the parallel speedup intrinsic to running the AML on a distributed set of nodes.

In Section 4.2.1 we showed that a WaveScript program can be written to adaptively process data locally or centrally, depending on certain conditions, with the intent of lowering the overall localization latency by reducing network costs. While this is an interesting idea, the conditions under which it is beneficial to process the data locally depend on whether the speedup in network transmission balances out the potential increase in latency due to local processing.

To evaluate this, we would ideally compare the goodput transfers of 32 KB and 800 byte data transfers in an identical network. Unfortunately, our 32 KB dataset in the multi-hop topology was gathered opportunistically, and we failed to get a corresponding 800 byte data during this period.

However, we did gather measurements of 800 byte transfers in a 1-hop topology. We use this data to get a rough estimate of the trade off between computing the AML algorithm locally or at the sink, by estimating the expected latency. In 154 transfers from seven nodes, the mean transfer time was 0.0212s (min 0.008s, max 0.0607s). Based on conservative estimates of packet loss and TCP timer behavior, we can use these results to formulate an estimate of the multihop performance.

We model the expected latency *E(x)* of an 800 byte transfer using the equation shown in formula 2. Since the MTU is 1500 bytes, our payload will easily fit into a single packet, therefore we need only model the expected latency of a single segment being sent in an established TCP stream, rather than a more complex TCP model.

We assume an exponential backoff starting at 1 second for our TCP retransmit timer, and a conservative estimate of packet loss probability P of 1/50, independent for each hop that a packet must travel over. Our expected loss rate for N hops is

$$P(N) = 1 - (1 - P)^N \quad (1)$$

and, using the formula for a geometric series the expected latency *E(x)* (assuming a mean latency per hop $H = 0.0212$) is

$$E(x) = N \cdot H + P(N)/(1 - 2 \cdot P(N)) \quad (2)$$

The results of this comparison are shown in Figure 11. We see that the measurements of the delays incurred by the 32 KB transfers scale approximately linearly with the number of hops, as we saw in Figure 10. In contrast, the single packet transfers, even accounting for packet loss and retransmission, arrive with very low latency, but incur the fixed 2 second latency of the AML computation. Incidentally, we note our 1/50 packet loss probability is conservative—the in-situ data collected from our 1-hop network didn't see even one backoff retransmission in 150

sends. A repeated test in a similar outdoor two-hop network saw just 6 retransmissions in 700 sends.

From this particular graph we conclude that there is a definite trade-off point for adaptation of processing at between 2 and 3 hops. At this point, a node would substantially benefit from performing its processing locally. We intend to investigate this trade-off more fully is subsequent work, as we feel it is an compelling problem.

6. Conclusion

In this paper we have described VoxNet, a platform for acoustics research, and applied it to a specific bioacoustics application. We described the architecture of the hardware and software components of the platform, and validated our approach by comparing against previous related work. We showed that in using WaveScript as our language of choice, our deployed application can consume less CPU and memory resources and provide more functionality, such as spill to disk. These factors are important for the users of our platform, as we want to enable high-level programming and rich interactions with the network, but without loss of raw performance. We believe VoxNet has a general appeal to a variety of high-frequency sensing applications, and in particular, will lower the barrier of entry for bioacoustics research.

References

[1] Labview. http://www.ni.com/labview/.

[2] The mathworks. http://www.mathworks.com.

[3] A. M. Ali, T. Collier, L. Girod, K. Yao, C. Taylor, and D. T. Blumstein. An empirical study of acoustic source localization. In *IPSN '07*, 2007.

[4] M. Allen, L. Girod, and D. Estrin. Acoustic laptops as a research enabler. In *EmNets '07*, 2007.

[5] A. Arora et al. Exscal: Elements of an extreme scale wireless sensor network. In *IEEE RTCSA*. IEEE, 2005.

[6] D. A. Driscoll. Counts of calling males as estimates of population size in the endangered frogs geocrinia alba and g. vitellina. *Journal of Herpetology*, 32(4):475–481, 1998.

[7] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *OSDI*, pages 147–163, Boston, MA, December 2002.

[8] A. Fillinger et al. Nist smart data flow system ii: speaker localization. In *IPSN '07*, 2007.

[9] J. C. George, J. Zeh, R. Suydam, and C. Clark. Abundance and population trend (1978-2001) of the western arctic bowhead whales surveyed near barrow, alaska. *Marine Mammal Science*, 20:755–773, 2004.

[10] L. Girod et al. The case for WaveScope: A signal-oriented data stream management system (position paper). In *Proceedings of CIDR07*, 2007.

[11] L. Girod et al. Emstar: a software environment for developing and deploying heterogeneous sensor-actuator networks. *ACM Transactions on Sensor Networks*, August 2007.

[12] L. Girod et al. Xstream: A signal-oriented data stream management system. In *Proceedings of ICDE08*, 2008.

[13] L. Girod, M. Lukac, V. Trifa, and D. Estrin. The design and implementation of a self-calibrating distributed acoustic sensing platform. In *ACM SenSys*, Boulder, CO, Nov 2006.

[14] O. Gnawali et al. The tenet architecture for tiered sensor networks. In *Sensys '06*, 2006.

[15] B. Greenstein et al. Capturing high-frequency phenomena using a bandwidth-limited sensor network. In *SenSys '06*, 2006.

[16] K. A. Hobson, R. S. Rempel, H. Greenwood, B. Turnbull, and S. L. V. Wilgenburg. Acoustic surveys of birds using electronic recordings: New potential from an omnidirectional microphone system. *Wildlife Society Bulletin*, 30(3):709–720, 2002.

[17] W. Hu et al. The design and evaluation of a hybrid sensor network for cane-toad monitoring. In *IPSN 2005*.

[18] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04*, 2004.

[19] C. T. Inc. Mica2 wireless measurement system datasheet, http://www.xbow.com/products/product_pdf_files /datasheets /wireless/6020-0042-03_a_mica2.pdf.

[20] Intel. Intel stargate, http://platformx.sourceforge.net, 2002.

[21] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[22] R. Karp, J. Elson, D. Estrin, and S. Schenker. Optimal and global time synchronization in sensornets. Technical Report CENS-0012, Center for Embedded Networked Sensing, 2003.

[23] A. Ledeczi et al. Countersniper system for urban warfare. *ACM Transactions on Sensor Networks*, (2):153–177, Nov. 2005.

[24] P. Levis and D. Culler. Mate: a Tiny Virtual Machine for Sensor Networks. In *ASPLOS-X*, 2002.

[25] J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao. Distributed state representation for tracking problems in sensor networks. In *IPSN '04*, 2004.

[26] P. K. McGregor, T. M. Peake, and G. Gilbert. Communication behavior and conservation. In L. M. Gosling and W. J. Sutherland, editors, *Behaviour and conservation*, pages 261–280. Cambridge: Cambridge University Press, 2000.

[27] K. Payne, M. Thompson, and L. Kramer. Elephant calling patterns as indicators of group size and composition: The basis for an acoustic monitoring system. *African Journal of Ecology*, 41(1):99–107, 2003.

[28] P. Volgyesi, G. Balogh, A. Nadas, C. Nash, and A. Ledeczi. Shooter localization and weapon classification with soldier wearable networked sensors. Technical Report TR-07-802, January 2007.

[29] A. Waibel et al. Smart: the smart meeting room task at isl. In *ICASSP '03*, 2003.

[30] H. Wang and et al. Acoustic sensor networks for woodpecker localization. In *SPIE Conference on Advanced Signal Processing Algorithms, Architectures, and Implementations*, August 2005.