SenseTK: A Multimodal, Multimedia Sensor Networking Toolkit

Phillip Sitbon, Wu-Chi Feng, Nirupama Bulusu, Thanh Dang
{sitbon, wuchi, nbulusu, dangtx}@cs.pdx.edu

ABSTRACT

This paper describes the design and implementation of a multi-modal, multimedia capable sensor networking framework called *SenseTK*. SenseTK allows application writers to easily construct multi-modal, multimedia sensor networks that include both traditional scalar-based sensors as well as sensors capable of recording sound and video. The distinguishing features of such systems include the need to push application processing deep within the sensor network, the need to bridge extremely low power and low computation devices, and the need to distribute and manage such systems. This paper describes the design and implementation of SenseTK and provides several diverse examples to show the flexibility and unique aspects of SenseTK. Finally, we experimentally measure several aspects of SenseTK.

Keywords: Video sensing, video applications, video adaptation.

1. INTRODUCTION

Over the last half decade, sensor networking technologies have deployed and demonstrated for a range of scientific, industry, and military applications. For example, researchers at Intel have demonstrated a sensor system to do predictive maintenance on ship vessels as well as semiconductor plants [12]. As small computing devices continue to be developed, the ability to integrate more complex data types such as audio, images, and video are becoming possible.

Multimodal and multimedia-based sensor networks are defined by a diversity of sensing modes and data types. Multimodal sensing can involve both traditional scalar-based sensor technologies and more complex computing devices such as a video sensor. Multimodal also can imply the integration of mobile components such as robots into a static scalar-based sensor network. Multimedia sensor networks can combine audio, images, and video to provide sensing. Several examples have been demonstrated in the past such as the Great Duck Island experiment (which used images and scalar sensors), cane toad monitoring in Australia (which used audio signal processing to distinguish types of toads), and Panoptes (which used video on low-power sensor platforms).

There are a number of unique aspects of such multimodal and multimedia-based (MM) networks. First, as the data types become more complex, the processing of the data becomes necessarily more application specific. For example, in the cane toad monitoring example, the signal processing is based upon the cane toad's vocal signature. For video and image processing in such networks, the processing of the video will be even more specific to the application. For example, a video camera pointed at the highway may want to convert the video into the speed of cars on the highway or detect cars on the side of the road. In another application, the processing might be completely different. Second, sensor networks pose a many-to-one information implosion problem. Unlike video streaming systems which can deliver a single stream to many hosts through multicast, MM networks can potentially deliver many streams to a single client. Thus, processing and adaptation of the data needs to occur deep within the sensor network for optimal power management and scalability. Third, the diversity of computing devices means that programmatically and operationally, the system needs to be able to bridge a variety of hardware and software configurations. Finally, with the scale of such networks, the ability to easily program and retask a large number of sensors is necessary.

While some primitive MM networks exist today, they are characterized by extremely brittle software infrastructures. That is, changing the functionality of the system requires significant intervention on the user's behalf, potentially requiring significant modifications to the software as well. Second, such applications are typically written and optimized by the computer scientists that know the low level hardware and networking issues, rather than the application writer, who may not be a computer scientist at all. The key limitation today's MM networks is that there is no bridge between the application writers and the diverse low level sensor hardware. While multimedia toolkits have been proposed in the past, they are not well suited for the unique challenges posed by sensor networks.

In this paper, we describe the design and implementation of SenseTK, a toolkit that we have designed and implemented to help bridge the applications and the diversity of low-level hardware in sensor networking applications. With the MM networks in mind, the toolkit was designed with a number of goals including deployability, programmability, management, and retaskability. We will describe the basic systems architecture of SenseTK and provide several examples that integrate audio, images, video, and application-specific processing within the network. We also describe the management as well as storage and retrieval aspects of SenseTK.

The remainder of this paper is organized as follows. In the next section, we will describe some of the background and related work necessary for the paper. Section 3 further motivates SenseTK, describes the high-level design and implementation of the system, and describes several implementation algorithms. The latter will demonstrate the power and ease of programming MM networks that SenseTK provides. Section 4 shows some basic performance numbers for some of the example applications. Finally, we summarize our findings and provide directions for future research.

1.1. Contributions of this paper

We believe that SenseTK contributes a number of ideas and technologies to the state-of-the-art. First, we believe that this is the first flexible and extensible framework with which to bridge low-level sensors and applications. Second, the design of a low-level Python-based framework we call *Cascades* for the networking and message passing system within SenseTK allows application writers to abstract many of the low level details (such as mote communication) as well as other details outside the scope of the application writer interest (such as image and video compression algorithms) into a much more accessible package. Finally, we have demonstrated a number of applications that can be built out of the SenseTK toolkit.

2. BACKGROUND AND RELATED WORK

The focus of this paper spans a large number of areas in sensor networking and programming, video streaming, video adaptation, multimedia toolkit design, and sensor hardware. In this section, we review some of the background material most closely related to SenseTK. We can add other references if the reviewers feel it is important enough. We begin with sensor networking platforms and programming paradigms. We then focus on basic multimedia toolkits and multimedia middleware.

2.1. Sensor Networking Research

2.1.1. Hardware Platform Technologies

In the last half decade, a myriad of low-level sensor networking technologies have been developed. There are a number of UC Berkeley motes developed with varying capabilities for use in sensor networking applications, including the WeC, Rene, Dot, Mica, MicaZ, Telos and XSM sensors [14]. The main characteristics of these sensors are that they have small amounts of memory, have very limited processing capabilities, and low-power operation. Building on the Berkeley mote platform, there have been significant numbers of projects related to sensor networking technologies. Much of the work has focused on providing efficient routing mechanisms and minimizing the power used to transmit packets throughout the network (e.g. collision avoidance algorithms). While these sensors are used primarily for scalar sensing, some of these platforms can perform basic audio processing.

More recently, low-power imaging and video sensor are starting to be demonstrated. The Cyclops image sensor is a low-power image sensor that can be attached to a Crossbow Mica2 sensor [20]. The image sensor is capable of some computation and inference. For example, it can be used to capture color histograms of image characteristics. Because of its small size and limited capability, such a sensor is not intended for repeated image acquisition. Generally, an image sensor that is capable of capturing, compressing, and transmitting images at a rate greater than several images per second can be considered a video sensor. The Panoptes video sensor demonstrated the ability to capture, compress, and transmit video on low-power embedded devices [3]. While requiring several watts of power, the video sensor was capable of capturing video in near real-time.

The devices described in this section simply highlight the fact that video and imaging processing are becoming possible and that the diversity in hardware will probably continue, requiring the systems software to be able to deal with such diversity. We refer to the higher-powered video sensors as *Stargate-level* devices because the Crossbow Stargate is currently the most power efficient embedded device capable of real-time video compression and is made by the company that makes most of the motes being used by the research community.

2.1.2. Sensor Network Programming

Early research focused on programming local support for sensor devices. Directed Diffusion[10] proposed a distributed data-centric communications paradigm for sensor networks, wherein sinks express interests for data based on named attributes (that may include spatial context), and the matching sensor data is routed to the nearest sink. Envirotrack[1], on the other hand, provides abstractions for naming physical objects in the environment, that can be used for building tracking applications.

More recent work has focused on *macro-programming*, referring to the notion of programming the entire sensor network in a high-level language that specifies global behavior translated automatically into programs that run on individual sensor nodes. Early examples of this notion were TinyDB[24] and Cougar[2], which consider the sensor network as a database, and provide support for declarative queries for sensor data. This abstraction supports data collection in a sensor network, including data from user-specified geographic regions.

Welsh and Mainland propose an Abstract Regions framework [27] to support more sophisticated sensor networks, featuring distributed actuation and control. This framework provides a more generic family of spatial operators that capture local communication within regions of the network, and allows for addressing nodes by region, as well as sharing data within a region. Examples of regions include k-nearest neighbors, spanning tree rooted at a node, etc. Kairos[9] and Hood[29] are other examples of region based programming paradigms. IrisNet[6] provides a query-based framework for building wide-area Internet-scale sensor networks of higher-level devices, such as webcams. Although queries can scale to very large networks, the implicit assumption is that users will be able to specify the geographic descriptors for each sensor device connected to the Internet.

There are currently various software architectures available for interacting and communicating with sensor network devices, both with and without higher-capability platforms such as the Stargate Gateway device[21]. For example, the *Tiny Application Sensor Kit* (TASK)[23] uses TinyDB in its mote networks and DBMS systems on Stargate devices to provide a suite of data collection and monitoring tools; however, it does not provide tools for alternative sensor platforms or devices, nor does it allow for the creation of distributed applications on Stargate devices. EmStar [7], a system which provides an application interface for low-power Linux devices (such as Stargate and iPaq), is generally effective in some application areas but also relatively inflexible. The system is capable of running distributed heterogeneous sensing applications, but unlike Cascades operates primarily according to the UNIX application process model, where individual parts are implemented as separate system processes. This is an effective model, but limits applications from directly extending to platforms such as Windows without significant modification. This problem is further exacerbated by a proprietary IPC mechanism. Other toolkits, such as MoteLab[28] and SNACK[8] only focus development on TinyOS-based[25] devices. TinyDB is a good example of a TinyOS-based device application that extends onto client platforms as well, with the primary goal of data aggregation and collection.

2.2. Multimedia Middleware

Over the last decade there have been a large number of efforts focused on providing toolkit and middleware support for multimedia authoring, streaming, content management and database storage. Many of these toolkits provide much of the middleware but require a fairly experienced person to integrate the middleware into a usable application.

There are two notable toolkits that have focused on allowing users to build distributed multimedia applications. The Continuous Media Toolkit (CMT) allows users to create distributed streaming applications with basic control over the type of media that is being used [15]. CMT allows users to modify TCL/TK scripts in order to tailor the application to the user's requirements. The TCL scripts are similar in vein to the SenseTK scripts. However, given the more application specific processing, the SenseTK scripts tend to be slightly more data centric than the CMT scripts. The Open Mash system [16] is successor to the CMT toolkit and has a much richer set of primitives and application possibilities than the CMT toolkit. The toolkit is focused mostly on end-host application development. The main difference between Open Mash and SenseTK is that SenseTK necessarily has more of the details of the actual underlying system exposed to the application. That is, application writers need access to low-level scalar sensors. Furthermore, the types of interaction with devices is also more involved. It is expected that in large-scale deployments that a single node may process the data from a large number of nodes.

3. SenseTK – A TOOLKIT FOR BUILDING MULTIMODAL, MULTIMEDIA SENSOR NETWORKING APPLICATIONS

Before we describe SenseTK, there have been two notable sensor networking applications described in previous ACM Multimedia conferences that we should mention. In Multimedia 2003, Feng et. al describe a video-based sensor networking platform that they designed an implemented [3]. The sample application they describe is a video surveillance applications that allows users to view video events that have occurred. The processing happens at the video sensors and a buffering and adaptation algorithm is implemented in order to ensure that the system is scalable. In ACM Multimedia 2005, Kulkarni et. al describe a hierarchical video system [13]. The proposed system combines low-level video cameras with fairly low quality video and higher-level, better quality, pan-tilt-zoom camera capabilities. In this system, the low-level cameras can trigger the higher-level video cameras.

There are number of things to note about these applications. First and foremost, they are working examples of multimedia-based sensor networking technologies. Second, while they work for their targeted application, they are not retaskable. That is, the application is tightly coupled with the hardware, making even trivial functional changes potentially difficult. Third they are not really integrated with traditional sensor networking technologies such as the motes. Finally, the systems are not really scalable given the amount of human intervention required to set-up the systems.

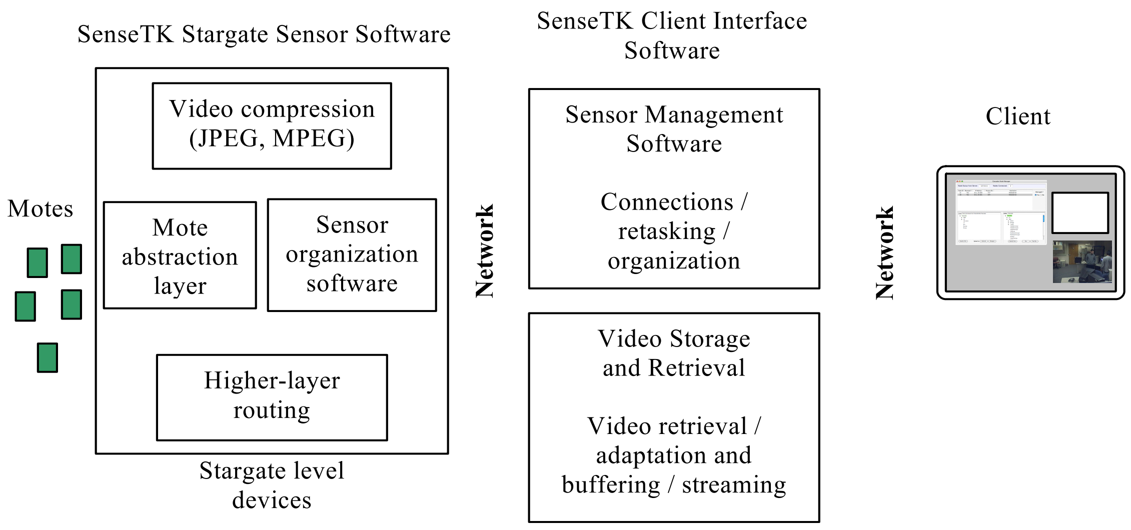

SenseTK Stargate Sensor Software
SenseTK Client Interface Software
Video compression (JPEG, MPEG)
Motes
Mote abstraction layer
Sensor organization software
Higher-layer routing
Stargate level devices
Network
Sensor Management Software
Connections / retasking / organization
Video Storage and Retrieval
Video retrieval / adaptation and buffering / streaming
Network
Client

Figure 1. This figure shows the overall architecture of the SenseTK toolkit.

3.1. Design of SenseTK

At a high-level, SenseTK has three major components (i) scalar and video sensor software, (ii) management and retasking, and (iii) storage and retrieval. The overall architecture is shown in Figure 1. The scalar and video sensor software runs on the Crossbow Stargate-level devices. The majority of the SenseTK toolkit consists of these components. It provides the bridge between the scalar sensors and the higher-level video sensors. The toolkit also consists of various compression algorithms and sensor network organization software. As will be described shortly, this consists of higher-layer routing primitives that allow a Stargate to refer to motes in an abstract way. We have also implemented a logical association framework that allow the sensors to automatically organize themselves into application meaningful ways by cross correlating scalar sensor and Stargate devices.

The management layer takes care of dealing with Stargate level scripts. As will be described shortly, this software allows a user to organize and group a number of stargate-level. It also allows a user to deliver updated scripts to all the sensors or a selected subset at a click of a button.

The storage and retrieval system allows a user to implement streaming retrieval applications that adaptively alter the video to fit within available bandwidth resources.

In the remainder of this subsection, we will describe the components in more detail. Readers interested in the scripts can skip to section 3.2.

3.1.1. Connecting and Programming Multimodal Sensors

The main contribution of SenseTK is the ability to create MM sensor networking applications with relatively small amount of code knowledge. The cornerstone of SenseTK is the ability to integrate mote sensing technologies directly into the MM application. SenseTK is built on top of a sensor networking application framework we have been developing that we call *Cascades*. Cascades provides a wide variety of organizational an networking tasks connected via Python scripts. SenseTK provides the framework around Cascades that makes it actually usable by application writers.

Python was chosen for the Cascades design implementation due to its relative flexibility and simplicity, as well as benefits of reduced complexity and a fast-paced development cycle[26]. Python follows the philosophy of creating applications with "batteries included," meaning that most applications can be written primarily from included modules/packages or freely available open-source modules/packages. Cascades can be considered an extension to this, since it essentially provides the "batteries" for sensor networking applications. Some very important tools (mentioned where they are used) were implemented in the Cascades system, keeping the actual Cascades code to a minimum.

The primary challenge in designing a system such as Cascades is making it robust enough to cover a wide variety of application-specific sensor networking needs (such as heterogeneity) without compromising simplicity or ease-of-use; the current state of the Cascades code base provides this. A secondary but equally important challenge is making sure that reusable code not only comes naturally from developing applications around the Cascades framework, but is also readily available for development and extension. While writing code in Cascades is conducive to the production of reusable components which we have written many of, the components listed here are limited to essential lower-level aspects.

With the aforementioned challenges in mind, designing an effective common paradigm among the base components becomes important. After investigating structured systems such as the Click router[11] which provides a strict push/pull data flow model, it became clear that a loosely defined structure would be beneficial, especially for the purpose of maintaining flexibility. The end result is what amounts to a function naming convention, where objects (i.e. modules or classes) can provide a `Get` function, which reads from an information source, and/or a `Put` function, which writes to an information destination. This very general design helps maintain the independence of objects, therefore increasing portability and modifiability. For example, an application may define a `MonitorCamera` module which monitors a video source for motion and reacts accordingly, requiring only an object with a `Get` function to retrieve video from. Suppose `Type1Camera` was used in an older project, and after some time a new implementation requires `MonitorCamera` again but needs to use a newer `Type2Camera`. With this design, `Type2Camera` can replace `Type1Camera` with no change to `MonitorCamera`'s logic. Any further abstractions, such as `MonitorCamera` requiring and object of type `Camera`, are left as application-specific details. The best part about this design is that it is completely optional, and extending it requires very little effort.

The following sections briefly discuss the important lower-level features of Cascades and their operation. The framework library is housed in a single Python package (which is really just a file system folder), consisting of "standard" packages, modules and functions. Separate from the library is the application modules, which are run with a utility script in a user-defined location. Figure 2 gives a high-level view of the basic networking structure.

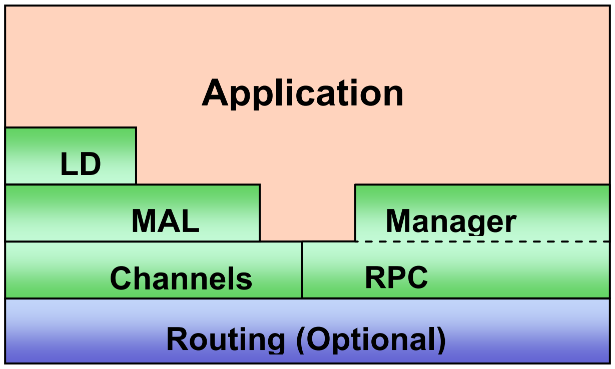


Figure 2. Cascades networking structure.

3.1.2. RPC

Remote procedure call mechanisms can have a wide variety of uses depending on the application. Cascades provides two RPC mechanisms: "native" RPC, which uses Python serialization for communication, and XML-RPC which uses a language-independent ASCII XML format. Each of the implementations has different advantages and drawbacks. Native RPC is fast and fully supports Python's fundamental types, but lacks interoperability with other programming languages – a feature that may be desirable for communication with other software packages. XML, on the other hand, is compatible between languages but exhibits much lower performance, especially on Stargate devices, and it also does not support all of Python's data types. An application that uses XML-RPC can be transparently converted to native RPC, but the opposite is not true because native RPC implementations might make use of data types not supported by XML-RPC. In either case, SenseTK has both, allowing application writer's to tailor the system to their applications.

3.1.3. Channel

A channel is defined in Cascades as a raw TCP/IP transport in which discrete items are sent and received. The underlying protocol is binary, simple, and commonly used. For each item, the sender sends the size of the data followed by the data itself. This module completely abstracts TCP/IP networking in such a way that the programmer needs to only either run a server or connect to one (both require no more than 3 lines of code), and subsequently communicate with the remote host.

3.1.4. Filter

A filter is an object that redirects a function call away from its intended target to one or more alternative functions. Because Python treats functions as variables, their destinations can be easily modified. Functionality for installation of an arbitrary number of *cascading* filter functions is provided, which can optionally discard the call to the original target. When calling the original target function, semantics remain unchanged and the filter functions are called transparently. With this module, filters can be arbitrarily complex, such as filters acting on filters, while the functions themselves remain relatively simple in nature.

3.1.5. Logging

Having an easy-to-use logging facility is very helpful for debugging purposes, as well as for general application monitoring. Logging events and information in Cascades is as simple as calling a function corresponding to the desired "log level" of the message. The log level can be configured to selectively report messages either to the screen, a file, or a remote log server. This can be configured by the application writer to be optimized for their particular application.

3.1.6. Logical Disassociation

One of the main challenges in heterogeneous sensor networking applications is providing a logical organization of the sensor nodes. To help solve this problem in a more scalable way, we have implemented a logical disassociation (LD) framework within SenseTK. That is, in deploying the sensor network, nodes for logical groups based upon network connectivity. The LD framework correlates phenomena within the sensor network, disassociating nodes that may be physically "connected" together but not logically meaningful. The LD package implements the concept of LD as reusable abstract modules which further extend a message handler by ignoring messages from disassociated sensors, which are identified through a training phase in which events are generated and correlated between sensors. Training and event classification modules are also provided.

3.1.7. TinyOS

Because TinyOS is the main operating system for scalar motes, we felt that it was important to provide access to low-level sensors through TinyOS. This component has four important modules, all of which assist communication with TinyOS-based sensor devices.

The Message module defines a TinyOS message object that contains general protocol-specific fields with error checking. It also defines a message handler object that abstracts sending and receiving messages according to the paradigm described above.

The serial module handles the (framed packet) communications protocol used to transmit TinyOS messages over a serial line. This is the heart of almost all communication with sensor devices because it is the easiest way to interface with motes and mote radio networks. It provides a handler which is able to multiplex incoming data to multiple waiting threads, as well as send messages with an option to require an acknowledgement with a timeout.

The Forward module is an extension to the Channel module that supports the communication protocol used in the SerialForwarder Java application included as part of the TinyOS distribution. It defines custom interfaces (that appear identical to the original) needed to complete the initial "version handshake" done by SerialForwarder. This module provides transparent connectivity with the existing tools for sending and receiving TinyOS messages over TCP/IP. The server requires only a generic message handler, so any message source can be forwarded (for example, serial or another forward). This module is mainly used by the SerialForwarder application module, which simply runs a forwarding server on the default port.

The MAL (Mote Abstraction Layer) module contains an interface to local or remote relaying services, as well as the service implementation itself, all of which support relaying of mote-level messages among higher-level devices. By using the MAL message handler, applications can access the extended capability of MAL without changes to existing code.

3.1.8. Video

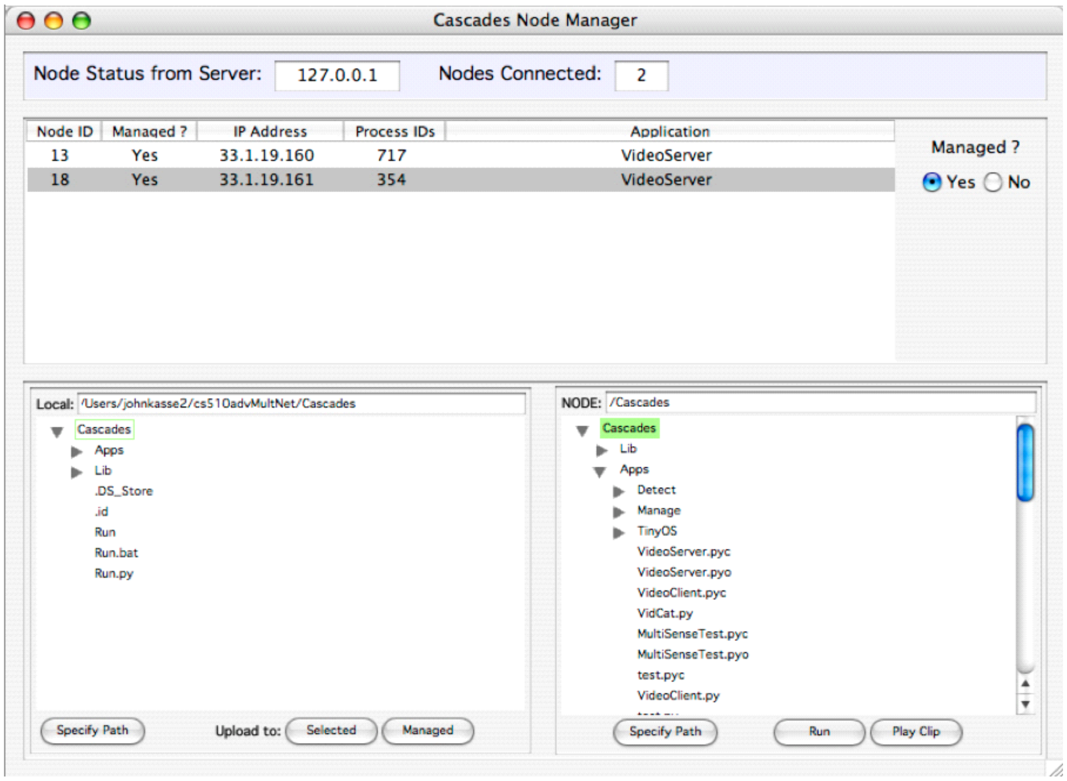
This package provides the basic mechanisms by which application writers can start to expand sensing into higher-level devices. It includes modules for capturing and encoding frames of video. Because several image and video compression algorithms have been provided by the community (e.g. DCT-optimized image processing from the Panoptes video sensor[3]), we have wrappered a JPEG module and and ffmpeg-based module for access to compression. These are especially useful for non-domain scientists wishing to implement MM applications.

The video capture module is written in C++, mostly for speed, and simply allows configuration of the device interface and capturing of frames with high-speed JPEG compression support. Video frames can be directly manipulated in Python, so any custom detection or processing algorithms are simple array operations. The encoder module was also written in C++ as a proxy to the functionality of the FFMpeg API[5] in order to provide high-speed MPEG video encoding.

3.1.9. Manager

The Manager module assists in increasing the programmability and retaskability of Cascades by providing an interface to distribute code and manage applications on several remote systems. With a graphical user interface, users can select applications to run, which nodes to run them on, and what parameters to pass to each. This application is an initial development of a system that could make programming for sensor system an *option* rather than a requirement, with the availability of previously developed code that can be "plugged in" to a custom application configuration. The image in Figure 3 shows the management user interface that allows nodes to be remotely controlled.

The top sub-window shows the stargate-level devices that are connected to the manager. The right side of the window shows the name of the script that is currently running on the stargate-level device. The managed button allows the user to "group" sensors


Cascades Node Manager

Node Status from Server: 127.0.0.1 Nodes Connected: 2

Node ID	Managed ?	IP Address	Process IDs	Application
13	Yes	33.1.19.160	717	VideoServer
18	Yes	33.1.19.161	354	VideoServer

Managed ? [✓ Yes] No

Local: /Users/johnkasse2/cs510advMultNet/Cascades

```
▼ Cascades
    ▶ Apps
    ▶ Lib
      .DS_Store
      .id
      Run
      Run.bat
      Run.py
```

NODE: /Cascades

```
▼ Cascades
    ▶ Lib
    ▼ Apps
        ▶ Detect
        ▶ Manage
        ▶ TinyOS
          VideoServer.pyc
          VideoServer.pyo
          VideoClient.pyc
          VidCat.py
          MultiSenseTest.pyc
          MultiSenseTest.pyo
          test.pyc
          VideoClient.py
```

Figure 3. This figure shows the basic management interface in SenseTK.

together that need to have the same scripts deployed on them. The lower left window show the directory structure of the local file system, which is machine independent thanks to the Python interpreter being machine independent. The lower right window shows the directory structure on the remote stargate. The manager assumes that the code to be executed resides in a static location.

3.1.10. Storage and Retrieval System

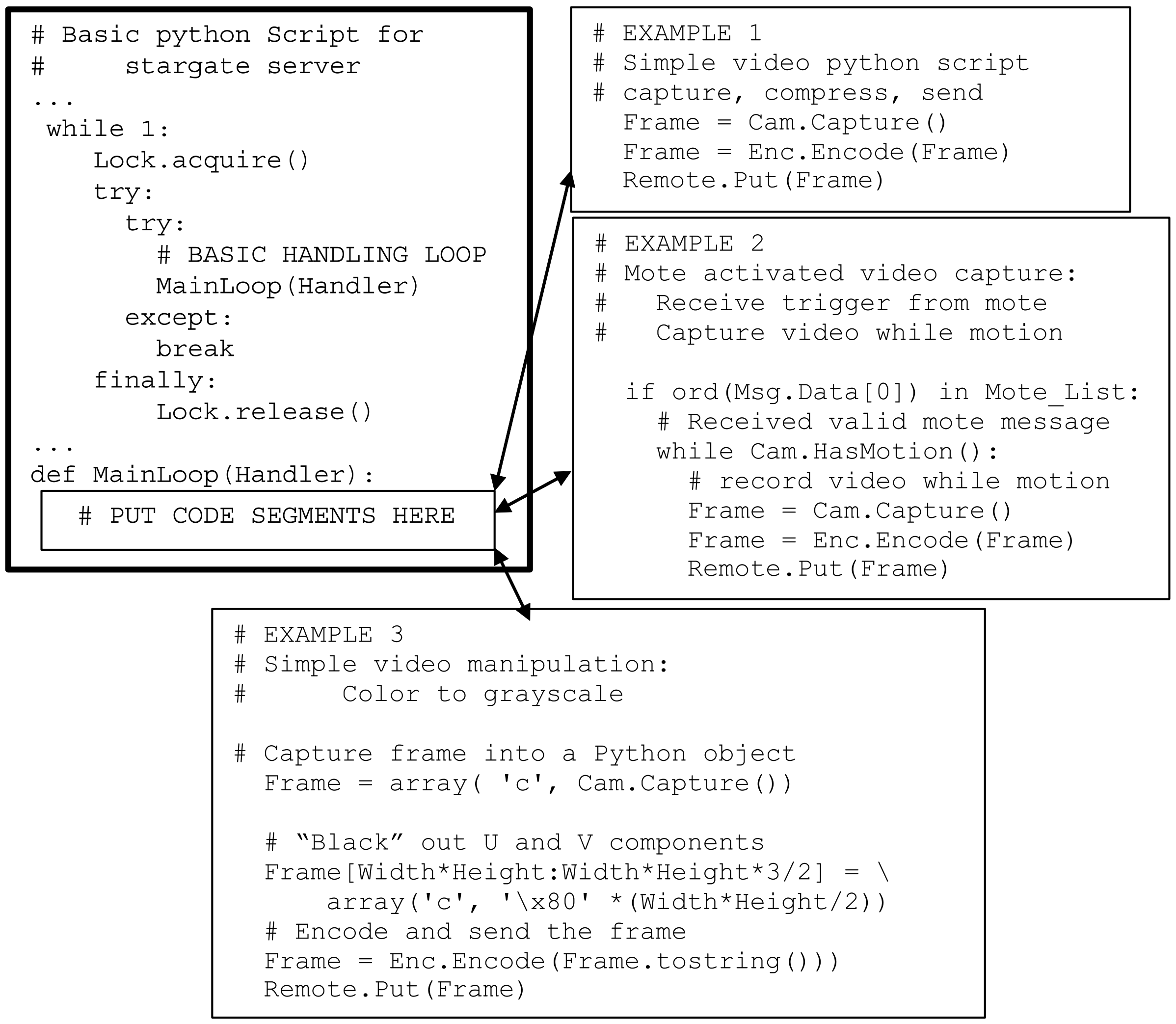
We have also implemented a storage and retrieval mechanism for the clients. The clients can access the video data remotely and have video displayed in a network-adaptive way. The buffering and adaptation algorithm is similar to that proposed by Feng [4]. The buffering and adaptation algorithm can be used within the sensor software as well. Because this software is similar to existing systems, we simply note that it exists and focus on the sensor specific code in the rest of the paper.

3.2. Example Applications

The difficulty in demonstrating the utility of carefully constructed toolkits is the low-level details get hidden and that many, many examples are needed in order to understand the impact of such a toolkit. That is, the power of the toolkit is in *what's missing*, rather than how complicated the programming can be. In this section, we will describe several really simple applications as well as some more complex application-specific applications that we have implemented.

3.2.1. Simple MM Sensor Networking Examples

In this subsection, we will provide three relatively simple examples of applications built using SenseTK. As previously mentioned, each stargate-class device in the network can have a number of motes connected to it and can optionally have a video camera attached to it. Each Stargate also runs a Python interpreter, which interprets the script that it needs to run. The wrapper functions of our scripts allow for automatic reloading of scripts that have updated. Thus, updating or changing the functionality of the system is a simple as replacing the script that it is running. Figure 4 shows three simple example video sensor code examples that we have used in our demonstrations. The bold outlined box shows the basic wrapper function of a video-based



```
# Basic python Script for
#     stargate server
...
 while 1:
    Lock.acquire()
    try:
      try:
        # BASIC HANDLING LOOP
        MainLoop(Handler)
      except:
        break
    finally:
        Lock.release()
...
def MainLoop(Handler):
   # PUT CODE SEGMENTS HERE
```

```
# EXAMPLE 1
# Simple video python script
# capture, compress, send
  Frame = Cam.Capture()
  Frame = Enc.Encode(Frame)
  Remote.Put(Frame)
```

```
# EXAMPLE 2
# Mote activated video capture:
#   Receive trigger from mote
#   Capture video while motion

  if ord(Msg.Data[0]) in Mote_List:
    # Received valid mote message
    while Cam.HasMotion():
      # record video while motion
      Frame = Cam.Capture()
      Frame = Enc.Encode(Frame)
      Remote.Put(Frame)
```

```
# EXAMPLE 3
# Simple video manipulation:
#      Color to grayscale

# Capture frame into a Python object
  Frame = array( 'c', Cam.Capture())

  # "Black" out U and V components
  Frame[Width*Height:Width*Height*3/2] = \
      array('c', '\x80' *(Width*Height/2))
  # Encode and send the frame
  Frame = Enc.Encode(Frame.tostring()))
  Remote.Put(Frame)
```

Figure 4. Examples of basic multi-modal, multimedia sensing applications in SenseTK

interconnect that we have implemented in SenseTK. The key here is that the *MainLoop* function is called once per time unit[1] and can do any number of things, including the three example code segments shown in the figure.

In example 1 of Figure 4, the script simply captures a frame of video from the camera, compresses the frame, and the sends it across the network. In this example, we have used JPEG compression. We have JPEG and MPEG compression implemented in the toolkit. MPEG is provided through a wrapper function that we have implemented for *FFMpeg*. Thus, changes to FFMpeg can be easily re-incorporated into the running system.

In example 2 of Figure 4, we show how easy it is to use motes to trigger the video sensor. In this case, the first line of code checks whether the mote is on the list of triggers - if it is, the camera begins recording video and continues to do so while motion is detected. We have implemented it such that the trigger line can also be attached to a single mote as well. The first byte of a mote message contains the mote ID, allowing the application to selectively listen or drop messages from motes that it is interested or not interested in, respectively.

The final example of Figure 4 shows how a more advanced user can access the camera data directly and perform manipulations on it. In this example, the data coming from the camera is in the YUV color space standard in many image and video representations. The YUV color space represents luminosity (grayscale) and two chrominance components. The example shown then removes the U and V colors so that the image sent across the network is then only a grayscale image. For

[1] The time unit here is up to the writer's discretion. It can be once every frame or can be called once for a number of frames.

deployments such as habitat monitoring, algorithms for detecting a particular species can be included either in the script or can call a module compiled in C that performs the detection.

Note that in all these examples, many of the complicated details such as image compression can be hidden from the user, if so desired. Thus, SenseTK can alleviate the implementer, who may not be a computer scientist, to take advantage of pre-defined and optimized modules. One unanticipated benefit of our infrastructure is that we have found it incredibly easy to test and deploy new applications that involve motes, sound, and imaging sensors. Changes involve simply changing the script and saving it. Errors manifest themselves fairly quickly, especially when video processing is involved, which, in turn, allowed us to debug the code on-the-fly. While it should not be used to replace good engineering practice, we have found it to be beneficial nevertheless.

3.2.2. Integrating computer vision algorithms into SenseTK

As an example of a more complex integration of code into SenseTK, we will describe a simple computer vision procesin algorithm integrated into SenseTK. Optical flow is a computer vision algorithm used to analyze the motion of moving objects. It has been used widely in tracking and robotics. In Figure 6, we use optical flow to analyze the captured frames to get the object location and velocity. In this example, the optical flow triggers video cameras to its left and right. This allows video from correlated video sensors to be grouped in an application meaniful way.

Figure 5 shows a face detection example. This code uses machine learning to detect the regions of the image which contain human faces. In the example, the algorithm calls CvDetectFace, which is a SenseTK wrapper for the Intel Open Source Computer Vision Library algorithm that detects faces within video. For this paper, we assumed the existence of a trained neural network. Our experiments will show that the Stargate can process such algorithms. Because of the processing speed, the buffering and adaptation system can be placed between the video capture and the face detection algorithm in order deal with bursty video sensor sources and the slower computer vision algorithms.

```
while True:
  Frame = Capture()
  # Returns cropped image of face, if detected
  Frame = CvDetectFace(Frame)
  if Frame:
    break

# Face detected - encode & send it
Data = Encode(Frame)
Put(Data)
```

Figure 5. A simple use of the face detection algorithm. Here, each frame is analyzed and the first frame containing a face is sent to the client.

```
while True:
  # Capture some frames
  Frames = Capture(30)

  # Use OpenCV to detect motion vectors
  Vectors = CvFlowAnalysis(Frames)

  # Returns vectors, i.e. ((x, y), (vx, vy))
  # Process the information, determine
  #   average direction of objects
  Direction = ProcessVectors(Vectors)
  # Will not be set if threshold condition
  #   not met.

  # Trigger adjacent cameras, which will
  # do the same analysis being done here.
  if Direction | DIR_LEFT:
    TriggerNext_Left()
  elif Direction | DIR_RIGHT:
    TriggerNext_Right()
```

Figure 6. Optical flow analysis example. OpenCV determines motion vectors for certain pixel groups and the app uses them to trigger other cameras.

3.2.3. Integrating audio into SenseTK

Because support for audio sampling is included, any available audio source can be integrated into an application within SenseTK. For example, MicaZ motes can sample audio from the standard sensor board because it includes a microphone. This data can be sent to a nearby Stargate for analysis, encoding, or triggering (as shown in Figure 7). The FFMpeg interface module also provides access to its audio codecs, allowing for seamless encoding integration.

```
# Record 100,000 samples of audio from mote
Count = 100000
# Get the frame size in samples
FrameSize = Info.FrameSize

while Count:
  Msg = Handler.Get()
  if not Msg or Msg.Source != AudioSource:
    # Invalid message
    continue

  # Save samples (one byte each)
  Samples += len(Msg.Data)
  Count -= len(Msg.Data)

  # Encode frames
  while len(Samples) >= FrameSize:
    Encoded = EncodeAudio(Samples[:FrameSize])
    Samples = Samples[FrameSize:]

# Store the data (ignores leftover samples)
file('saved.mp2', 'wb').write(Encoded)
```

Figure 7. Example audio application. Audio data is received from a mote and encoded in frames of samples. For recording, no further processing is required. Because the data is raw, it can be passed to a FFT function for analysis or signal detection if necessary.

4. EXPERIMENTATION

Python is written in highly optimized C code, so it is usually fast for most reasonable applications. Anecdotal evidence supports the performance capabilities of Python for a wide variety of applications, as well as roughly a substantial increase in performance over version 2.2, the previous major revision with significant differences. One of Python's greatest strengths lies in its extensibility through C/C++ modules – therefore, any Cascades code that does or might cause a performance strain can be written in C/C++. Python is lacking in performance with respect to some noticeable specifics, such as multithreading and average function call overhead time (which is *relatively* poor but not crippling). Fortunately, profiling and optimizing Python code is also a trivial task, thus alleviating most, if not all bottlenecks. In this section, we compare the performance of similar applications written in Python and C. We also inspect the performance of various computer vision algorithms (implemented primarily in C, but with some Python wrappers) when run on a stargate device.

Figure 8 compares the performance of Python and C for three different scenarios on a stargate device. The first scenario is a direct video capture (no data saved), which is synonymous to reading a file descriptor in Linux, where the Python code calls into a C module to capture video and the C code directly captures it. Because the capture code in C is the exact same for both applications, the overhead shown is purely the cost of running the Python code and calling into the C module. The second row, however, includes a conversion to grayscale that manipulates the contents of each frame in Python and C, respectively. It shows a situation where video data can be directly manipulated in Python without a large cost, although the algorithm is rather simple. The third row depicts the performance when each frame is being compared to the last – this case has more extensive image manipulation, but does not perform too poorly.

	160x120		320x240	
	Python	C	Python	C
Capture	29.47 fps	29.64 fps	16.48 fps	16.58 fps
Process	28.95 fps	29.39 fps	16.02 fps	16.35 fps
Difference	28.08 fps	28.25 fps	15.80 fps	15.88 fps

Figure 8. Performance of Python vs. C for video processing on a stargate.

Figure 9 show the performance of some OpenCV algorithms run on a Stargate device. The statistics for capturing and saving to disk are different from above, since OpenCV has its own mechanism for both. The optical flow algorithm picks a user-defined number of points that have the highest probability to be in motion within the image. The larger the number, the longer the computation takes. For the results given, we chose 100 points, a reasonable choice for 160x120 video. The running times for optical flow and face detection are not very impressive at the moment; optimization did not yield any better results, largely due to the hardware limitations. As mentioned in the previous section, adaptive buffering mechanisms can alleviate the performance problems by processing events when no other events are occurring, i.e. during idle time. This would especially help in the case of the last experiment, which uses a face detection algorithm (a trained neural network) on each frame to determine a region of interest within the image.

	160x120	320x240
Capture	29.5 fps	16.5 fps
Capture + Save	4.89 fps	1.89 fps
BG Differencing	6.17 fps	6.11 fps
Optical Flow	0.11 fps	0.06 fps
Face Detection	0.06 fps	0.01 fps

Figure 9. OpenCV algorithm performance on the Stargate (100 feature points for optical flow)

Figure 10 depicts the bandwidth difference between normal (full frame) capturing versus cropped (face only) capturing. In the cropped capturing experiments, the video is analyzed and sent only if the presence of an object is detected. Furthermore, it only transmits the region that has the event (Region of Interest, or ROI) and ignores all other surroundings. Hence, it saves bandwidth by limiting transmission to detected events, and creates further savings by sending only the important part of the image (the face).

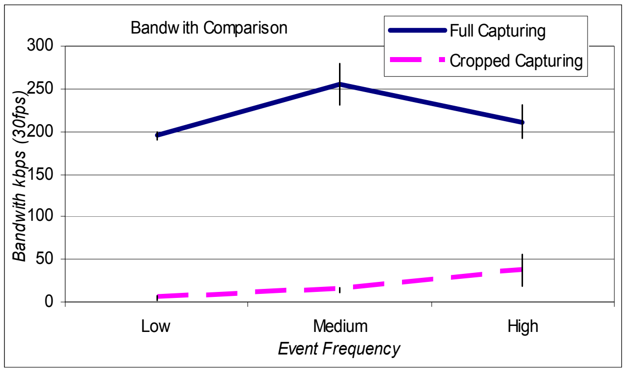


Figure 10. Full image capturing vs event based face recognition capturing.

5. CONCLUSION

In this paper, we have described the design and implementation of SenseTK, a multi-modal and multimedia-based sensor networking framework. We showed how SenseTK allows application writers to easily construct multi-modal, multimedia sensor networks that include both traditional scalar-based sensors as well as sensors capable of recording sound and video. We also described the design and implementation of SenseTK and provided several diverse examples to show the flexibility and unique aspects of the system. Finally, we measured several aspects of SenseTK and showed its ability to perform well. We hope that this framework will become a well-known tool for multimedia sensor networks. To aid more common use, we hope to develop a large base of reusable modules that will further assist the creation of diverse and innovative applications.

6. REFERENCES

[1] T. F. Abdelzaher, B. M. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, Lin Gu, Tian He, Sudha Krishnamurthy, Liqian Luo, Sang Hyuk Son, Jack Stankovic, Radu Stoleru, Anthony D. Wood: "EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks." ICDCS 2004: 582-589.

[2] P. Bonnet, J. E. Gehrke, and P. Seshadri. Querying the Physical World. *IEEE Personal Communications*, Vol. 7, No. 5, October 2000, pages 10-15. Special Issue on Smart Spaces and Environments.

[3] Wu-chi Feng, Brian Code, Ed Kaiser, Mike Shea, Wu-chang Feng, Louis Bavoil, "Panoptes: A Scalable Architecture for Video Sensor Networking Applications", in *Proceedings of ACM Multimedia 2003*, Nov. 2003, pp. 562-571.

[4] Wu-chi Feng, M. Liu, B. Krishnaswami, A. Prabhudev, "A Priority-Based Technique for the Best-Effort Delivery of Stored Video", in *SPIE/IS&T Multimedia Computing and Networking Conference*, San Jose, California, pp. 286-300, January 1999.

[5] "FFmpeg Multimedia System." The FFmpeg Project. 20 May. 2005. SourceForge. 12 Jun. 2005 <http://ffmpeg.sourceforge.net/index.php>.

[6] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, S. Seshan IrisNet: An Architecture for a World-Wide Sensor Web *IEEE Pervasive Computing, Volume 2, Number 4 (October-December 2003)*

[7] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, D. Estrin., "Emstar: A software environment for developing and deploying wireless sensor networks", in *Proc. of the 2004 USENIX Tech. Conf.*, Boston, MA, 2004. USENIX Association.

[8] B. Greenstein, E. Kohler , D. Estrin, "A sensor network application construction kit (SNACK)", in *Proc. of the 2nd international conference on Embedded networked sensor systems*, Baltimore, MD, November 03-05, 2004.

[9] R. Gummadi, O. Gnawali, R. Govindan, "Macro-programming Wireless Sensor Networks using Kairos," in *Proc. of the International Conference on Distributed Computing in Sensor Systems (DCOSS),* June 2005.

[10] J. Heidemann,, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, Deepak Ganesan, "Building efficient wireless sensor networks with low-level naming", Proc. Of ACM SOSP 2001.

[11] E. Kohler, "The Click Modular Router", *PhD thesis*, MIT, Feb. 2001.

[12] L. Krishnamurthy, R. Adler, P. Buonadonna, et. al, "Design and Deployment of Industrial Sensor Networks: Experiences from a Semiconductor Plant and the North Sea", in *Proc. of the ACM 3rd International Conference on Embedded Networked Sensor Systems*, Nov. 2005, pp. 64-75.

[13] Purushottam Kulkarni, Deepak Ganesan, Prashant Shenoy, Qifeng Lu, "SensEye: A Multi-tier Camera Sensor Network, in *Proceedings of ACM Multimedia 2005,* Nov. 2005.

[14] J. Polastre, Design and Implementation of Wireless Sensor Networks for Habitat Monitoring, Master's Thesis, Dept. of EECS, University of California Berkeley, Spring 2003.

[15] Ketan Mayer-Patel, Larry Rowe, "Design and Performance of the Berkeley Continuous Media Toolkit", in *Proceedings of SPIE Multimedia Computing and Networking 1997*, Proc. SPIE 3020, pp 194-206.

[16] S. McCanne, et al. "Toward a Common Infrastructure for Multimedia Networking Middleware", in *Proceedings of NOSSDAV 97*, St. Louis, Missouri, August 1997.

[17] "MICAz ZigBee Series (MPR2400)." Crossbow Technology. 12 Jun. 2005 <http://www.xbow.com/Products/productsdetails.aspx?sid=101>.

[18] "Multi-Sensor Module (MTS300/310)." Crossbow Technology. 12 Jun. 2005 <http://www.xbow.com/Products/productsdetails.aspx?sid=75>.

[19] Python. 12 Jun. 2005 <http://python.org>.

[20] Mohammad Rahimi, Deborah Estrin, Rick Baer, Henry Uyeno, Jay Warrior, "Demo Abstract: Cyclops: image sensing and interpretation in wireless networks", *in Proceedings of ACM SenSYS 2004*, Baltimore, November 2004.

[21] "Stargate Gateway (SPB400)." Crossbow Technology. 12 Jun. 2005 <http://www.xbow.com/Products/productsdetails.aspx?sid=85>.

[22] "Serial Gateway (MIB510)." Crossbow Technology. 12 Jun. 2005 <http://www.xbow.com/Products/productsdetails.aspx?sid=75>.

[23] "Tiny Application Sensor Kit (TASK)." Intel Research laboratory at Berkeley. 26 Sep. 2003. Intel Research, Berkeley. 12 Jun. 2005 <http://berkeley.intel-research.net/task/>.

[24] "TinyDB: A Declarative Database forSensor Networks." 12 Jun. 2005 <http://telegraph.cs.berkeley.edu/tinydb/>.

[25] TinyOS. UC Berkekey. 12 Jun. 2005 <http://tinyos.net>.

[26] Venners, Bill. "Programming at Python Speed." Artima Developer. 27 Jan. 2003. Artima Software. 12 Jun. 2005 <http://www.artima.com/intv/speed.html>.

[27] Matt Welsh and Geoff Mainland, "Programming sensor networks using abstract regions", Proc. NSDI 2004, pp. 29—42.

[28] G. Werner-Allen, P. Swieskowski, and M. Welsh, "Motelab: A wireless sensor network testbed", in Proc. Fourth International Conference on Information Processing in Sensor Networks (IPSN'05), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS), April 2005.

[29] K. Whitehouse, C. Sharp, D. E. Culler, E. A. Brewer: "Hood: A Neighborhood Abstraction for Sensor Networks." MobiSys 2004.