7-31-2019

Implementation and analysis of the ISO/IEC/IEEE P21451-1 draft standard for a smart transducer interface common network services and its applications in the Internet of Things

Russell Henry Albert Trafford
Rowan University, traffo17@students.rowan.edu

Recommended Citation

Trafford, Russell Henry Albert, "Implementation and analysis of the ISO/IEC/IEEE P21451-1 draft standard for a smart transducer interface common network services and its applications in the Internet of Things" (2019). *Theses and Dissertations*. 2725.
https://rdw.rowan.edu/etd/2725

IMPLEMENTATION AND ANALYSIS OF THE ISO/IEC/IEEE P21451-1 DRAFT STANDARD FOR A SMART TRANSDUCER INTERFACE COMMON NETWORK SERVICES AND ITS APPLICATIONS IN THE INTERNET OF THINGS

by

Russell Henry Albert Trafford

A Thesis

Submitted to the
Department of Electrical and Computer Engineering
College of Engineering
In partial fulfillment of the requirement
For the degree of
Master of Science in Electrical and Computer Engineering
at
Rowan University
August 31, 2016

Thesis Chair: Dr. John L. Schmalzel

Abstract

Russell Henry Albert Trafford
IMPLEMENTATION AND ANAYLSIS OF THE ISO/IEC/IEEE P21451-1 DRAFT STANDARD FOR A SMART TRANSDUCER INTERFACE COMMON NETWORK SERVICES AND ITS APPLICATIONS IN THE INTERNET OF THINGS
Spring 2019
John L. Schmalzel, Ph. D., P.E.
Master of Science in Electrical and Computer Engineering

The Internet of Things (IoT) has rapidly become the paradigm for the creation and improvement of new and old Cyber Physical Systems (CPS), but how much longer can this development of IoT devices, networks, and services be sustained? The past decade has seen incredible growth in internet connected devices, with current estimates placing the number of such devices at about 20 billion in 2017, not including personal computers, smart phones, and tablets. Since these new and emerging markets are competitive, there originally was no incentive to design systems, which were built to have a common protocol to enable interoperability between systems. This can pose a large integration effort if two or more of these systems need to communicate together as part of a larger system. The revitalization of the IEEE 1451 family of standards can solve this problem. The work in this thesis proposes to solve the integration problem by providing a common set of services and protocols for devices. This work provides the basis for a common architectural foundation for future IoT development. The contributions of this thesis include a renewal of the language and intent of the IEEE P21451-1 draft standard, development of example implementations to be included in the standard, and the development of Open Source hardware and software aimed at lowering the cost of adopting this standard.

Table of Contents

Table of Contents (Continued)

Table of Contents (Continued)

List of Figures

List of Figures (Continued)

List of Figures (Continued)

Chapter 1

Introduction

Introduction to the Internet of Things

Smart Cities, Smart Grids, Intelligent Factories, and Autonomous Vehicles are all areas in which billions of dollars in technological development and research are being spent. There are two main underlying technologies which are being used to power these areas which has come to be known as the Internet of Things (IoT). The first is the underlying sensor and transducer technology; the second, and rapidly becoming the most important, is the ability for sensors to communicate. From a business standpoint, this poses an immense opportunity to capitalize on these emerging fields. Not only does industry have an interest in these interconnected systems, the expansion of the maker movement has led to increased availability of commercial, off-the-shelf (COTS) systems. From a research standpoint, the ability to collect Big Data for deep learning applications, analyze system performance, and coordinate extremely large distributed systems require many new methods to be developed and tested.

The IoT is a paradigm, which addresses the fundamental change in the use of the Internet. IoT moves away from the original usage connecting people to other people or businesses to connecting "things." The most generic definition of a "thing" is anything that can connect to the Internet and can have an IP address assigned to it, thereby allowing it to send and receive data. Due to the availability and ease of access to the internet, objects such as cars, parking meters, industrial systems, and everything in between falls under this definition.[1]

The IoT can be broken into three main elements: sensing and communication; data storage and analytics; and user-level applications. The most significant areas of development are the user-level applications, due to the amount of money companies can make. Companies such as Nest Labs and Samsung have released platforms based on giving homeowners the ability to control their lighting, heating, kitchen appliances, and even their locks all from their smartphone. Some companies charge for these services whereas others include cloud services within the price of the device. There is also a market for building applications, which can tie together different companies' services together into one usable platform [1].

These applications are essentially a user interface to a server responsible for collecting data from multiple sources and analyzing trends within the data. An example of this is an office building with multiple tenants which contains a smart HVAC system [2] [3]. The overall system is controlled via a server which can query the thermostats in each room of the building and analyze whether any corrective conditioning needs to be done. If one of the tenants is going to have a conference of 20 people in their meeting room and want to make sure the room is comfortable once everyone is in the room. An office worker would have access to a thermostat or panel which they could set the desired temperature of the room. The system then begins to chill or heat the room to that temperature. As people start to walk into the room, the thermostat will register an increase in temperature and alert the main control system to adjust. This goes on throughout the meeting and as people leave.

The data collected from these thermostats can improve this is the system by collecting and analyzing the temperature pattern that emerged to maintain the correct temperature. This profile can be stored for later use in other parts of the building. This makes it where anyone else wishing to use the room for a meeting could pre-set the temperature of the room and the system could better cope with the change in occupancy. Using the knowledge of a future large occupancy, the system can pre-chill the room lower than the requested temperature so that by the time that all the persons (or heat sources) are in the room, the temperature of the room is at the desired level. This is just the surface of what an intelligent, smart building can do [2] [4].

It can be taken another step further by introducing occupancy sensors which traditionally have been used to detect whether a person is in a room. While sensors based on sound levels or passive infrared light can detect presence, they can not necessarily detect the number of people in a room. Using techniques such as facial recognition and radio frequency identification (RFID), the HVAC system could determine how many occupants are in a room and utilize this information to better condition the room. If this occupancy sensor network is sophisticated and distributed enough, it could even determine patterns and flows of persons within the building and begin to pre-condition rooms based on the time of day. Using micro-location techniques to pinpoint the exact locations of persons within a building can also aid in these patterns. With these patterns, the system could know, for example, that the conference rooms and other parts of the buildings are rarely occupied during the week. The system then could set the temperature higher or lower in these rooms based on the outside weather conditions to reduce the energy usage [5].

This Smart HVAC system is just one subsystem with the entire Smart Building, and this Smart Building is just one building within a potentially smart city. Each building within the city could have its own subsystems, and these buildings could communicate to one another in some applications. Buildings are just one part of a city. With systems such as traffic, lighting, utilities, air quality monitoring, pedestrian safety, and more, cities are complex with dynamic systems that need to communicate with multiple controllers at any given time. The underlying foundation of all these systems is the ability to sense, actuate, and communicate. The IoT provides a framework in which sensors and actuators from different manufactures can interoperate by communicating their information through gateways, brokers, concatenators, or other devices [6], [7].

IoT Networks and Fog Computing

By its previous definition, if every sensor and actuator were to be treated as a Thing, hundreds of new nodes which need to be addressed and handled by the building's network. This could potentially cause the building mangers to spend unnecessary funds in upgrading the network to handle all the new devices and traffic. However, does every sensor need to be able to communicate over the internet, or can some of these sensors be grouped together and treated as one single Thing? By grouping the sensors together, the number of devices that need addressing and the amount of traffic on the network can be reduced, while simultaneously still being able to be considered a Thing. Outside of the world of Smart Buildings, this is desirable for IoT networks relying on cellular networks to communicate over the internet back to some server. With cost being one of the driving factors in many decisions to adopt a new method, using this abstraction method could reduce the amount of hardware and data required to enable a system to be IoT capable [8].

This is the main idea in a new field of computing called “Fog Computing”. Taking its name from meteorology, Fog Computing aims to be the middleman between traditional local (ground) computing and the newer paradigm of cloud computing. Cloud computing is the idea that users can have reliable, on-demand access to many hardware and software services which traditionally would not be available due to hardware costs. These types of systems reside on the internet to make access easier; however, this limits both the cloud service and the consumer to the capabilities of their Internet Service Providers (ISP). A company may have the available hardware and networking support to provide a service such as cloud storage of video, but if the end-user has a low-bandwidth connection, the quality of the service will suffer. This also can be seen in a scenario where multiple users with high internet speeds simultaneously attempt to access the same file or program. While the user can support the amount of data transfer required to run the service, the cloud infrastructure may not be able to handle the requests at the same time.

With the prevalence of machine learning, deep learning, and artificial intelligence in most applications created today, there is also a problem with transferring that amount of data to a cloud service. Fog Computing also aims to reduce this traffic by extending the preprocessing, labeling, segmentation, or other bandwidth consuming task down to the devices generating the devices. In the example of the Smart Building posed above, a more traditional way of performing any learning on the data is to send it to a database in the cloud and use its resources to perform classification or training. As gateway or “edge” devices become more powerful and cheaper to implement, systems can now begin to do these tasks before reaching out to the cloud, reducing the bandwidth needed to connect to the cloud as well as reducing the number of requests sent to these servers.

These are examples of utilizing Fog Computing ideals to reduce the bandwidth of a smart sensor network, but so far there has been the assumption that the bandwidth is not limited. In more rural applications, there may not be easy access to high-speed or high-reliability internet connections. Fog Computing can still be beneficial in these areas to help reduce this requirement, however, another area which has been growing is remote sensing systems connected by cellular networks [9].

Low Bandwidth: Cellular Networks

The availability of Smart Phones and the popularity of the 4G networks provided in the United States has led to a sharp increase in the amount of data transmitted per user. While the cellular network companies are working constantly to support this traffic, what is going to happen when technologies such as automated vehicles which are time sensitive to commands and produce large amounts of sensor data come to the market? When a person is on Facebook or YouTube on their phone, some latency is expected in the form of buffering, but for an automated vehicle, having to buffer could mean making a life-critical decision on old data. As the IoT expands, it can only be expected as more sensor nodes join a network that there are going to be more of these latency issues. There needs to be a communication architecture developed which minimizes the amount of traffic added to these existing networks, while still providing users access to the transducer information they request.

Cellular network companies in the United States on average are beginning to drop support for older network architectures and coding schemas. Multiple third-party companies plan to use this to their advantage and market the slower, bandwidth-restricted networks towards the IoT [10], [11]. An example of this type of cellular connection can be seen with the Neo SIM Program. As of writing this thesis, if a network of sensors were to be implemented with 10 nodes, each with their own SIM card, and utilizing up to 750kB of data per month, it would cost $27.50 for the SIM cards themselves, and $10.00 a month for the data. There are additional costs which come in the form of hardware required to utilize the SIM card. Altogether, it would cost the manager of the network roughly $1 per node per month to keep the network running. Since most of these available programs charge for the amount of data utilized, it would be in the best interest of developers to work towards a network architecture which minimizes the amount of data needed to control the transducers [12].

To analyze where data limitations can be put in place, a model of the general architecture of a smart transducer network needs to be generated. At its core, most existing systems include a party which requests the information from or requests a change to a transducer module, which is connected directly to a sensor or actuator. This commanding party is commonly known as the Client will be referred to as such throughout the rest of this thesis. This Client can take the form of a variety of embodiments, such as a Smart Phone application, web page interface, control server, etc. An average consumer trying to control their lights in their house through the internet will most likely be using a smartphone application, whereas a large building HVAC system will most likely be using a control server that manages the temperatures in each of the rooms. For these systems to be able to work, however, all parties involved (whether directly or indirectly) must be able to talk and communicate over the internet. One interoperability problem is that there are many ways to communicate over the internet, each with its benefits and downfalls. With all the potential options available to developers, without a scaffold to build from, there will be a multitude of different ways to implement a network, which can lead to integration problems.

The Argument for Standardization

As the drive for connected cities and large-scale sensor networks grows stronger, so does the inherent need for a standardized approach to developing these networks. The issue with many projects created in the early days of the IoT is they were normally so narrow in scope. This led to decisions about the framework of the architecture which, at the time, seemed sufficient. However, these architectures quickly could run into problems once the companies or engineers tried to expand the available services. An example of this would be a company which started by developing thermostats which could be controlled from anywhere and could learn user behavior want to expand fully into the Smart Home market including devices such as door cameras. Originally the architecture for passing commands and data could be very simplistic for needing only single points of data, but now there needs to be support for streaming video. This is still only restricting this example to a singular company. Once the Smart Home market began to flourish, more companies with better or cheaper products may each use their own servers and protocols, making it very difficult to integrate with other companies. This is seen in almost any facet of the IoT, not just in Smart Homes.

At its core, most IoT systems are built for the simple task of retrieving sensor data or to control some transducer. Previous implementations of IoT systems varying from using cars within a city as central hubs for sensors to achieve a real-time flow of traffic, to controlling the temperature in a house, to even automated manufacturing systems revolve around these two fundamental functions [13]. It may seem like too much of a trivialization to just say this is all that is required; but everything that happens, between the request being sent over the internet to the physical sensor or transducer doing something, is all there to facilitate the ability to read or write to a transducer. This freedom is what gives power to designers to customize their own proprietary way to communicate to sensors. Without abstraction layers or some sort of structure, there is no guarantee that one person's internet capable device will be able to talk to someone else's. Without the ability for subsystems to communicate easily, innovation and further advancement at a higher-level grind to a halt.

By standardizing the architecture for the IoT and encapsulating specific responsibilities within layers of abstraction, designers and developers will have a foundation to build their systems from. As these networks are built, they will have at their core the ability to be interoperable with other systems, abstracting away the complexities of their own system to allow for simplistic integration into a larger distributed transducer scheme. There will be few people with the resources to be able to build a transducer network from the silicon level up to the application level, but most development into the IoT will come from the private sector trying to enter a brand-new market for their clients. These companies and corporations may only specialize in sensor modules, control boxes, or software platforms. It is vital to build this cyber-infrastructure in such a way that devices from different manufacturers have a simple way to communicate with one another. This can be accomplished through the IEEE 1451 family of Smart Transducer Network standards, allowing each abstraction layer to interact with one another with minimal knowledge about what is actually happening inside each layer [14], [15], [16], [17], [18].

IEEE 1451 Current Implementations

The architecture laid out in the IEEE 1451 family of Smart Transducer Network standards has been used in real-world applications spanning from traffic and air quality monitoring in a city to agricultural sensing networks to power generation and transmission. Kularatna and Sudantha focused on the generation of IEEE 1451 compliant gas sensor modules to measure concentrations of carbon monoxide, nitrogen oxides, sulfur oxides, and suspended particulate matter along with other toxic gasses. After comparing different sensor manufacturing techniques and discussing the necessary signal conditioning, they established an architecture to relay the sensor readings back to a client. Utilizing the Transducer Electronic Data Sheets (TEDS) as defined in the standard, information about each sensor connected to a module could be accessed by the client and is used in calculating any readings from the sensors. This information includes manufacturer identifiers, location information, as well as calibration dates and information. By utilizing the IEEE 1451 family of standards, they were able to create a network of these modules which allows for easy expansion to add new sensors types or more modules [4].

Kim et al [8] designed a Sensor-Ball based system for monitoring the health and transmission characteristics of high voltage power lines. This sensor-ball contained temperature sensors, wind direction, and velocity sensors, tilt sensors, a camera, and a GPS, along with batteries, charging circuitry, solar panels, CPU, and a ZigBee radio. Since these sensor-ball systems could be placed on many transmission lines, the issue of tracking the measurements, health, and calibration of each of these nodes became a daunting task, not to mention having the system work with multiple versions of the hardware. In their work, the team was designing the system to be IEC 61850 compliant; however, a common interface method for delivering this information was not found. For this, they turned specifically to IEEE 1451.0 and IEEE 1451.1 to supplement these missing functionalities. Higuera and Polo also had a similar task in their research by applying the 1451 standard to a 6LoWPAN network by creating a compact Transducer Electronic Datasheet [19].

Wei, et al., analyzed the functionality of IEEE 1451 standards in use with ISO 11783 to generate a complete set of plug-and-play capabilities for precision agricultural monitoring [20]. In their specific work, they applied the concept of a smart transducer and the Network Capable Application Processor (NCAP) to monitor weed growth, noting the effects of adding additional hardware to the transducer level for cost and functionality. Fernandes et al [12] continued this work to further develop into a common framework for precision agriculture and viticulture. Their team utilized a ZigBee based architecture for the Transducer Interface Module and a 1451.1 compliant NCAP as the gateway back to their database.

Bissi, et al. [1] and Kularatna and Sudantha [4] both take on the challenge of air quality monitoring and the detection of potentially harmful gasses. Both groups work focuses on the design of a TIM, which is based on gas sensors to detect concentrations of volatile organic compounds. The reason these researchers chose to adapt their technologies to the 1451 standard was the common interface that could be provided to multiple types of gas sensors while having the ability to be implemented at a relatively low cost. R. Wall and A. Huska worked towards generating a design platform for traffic signals which could be IEEE 1451 Compatible. The researchers focused on the Network

Capable Application Processor to Transducer connection, standardizing the way the traditional stoplight components connect. Utilizing this approach would allow for a designer to quickly choose between different components as well as simplify the software update process [21].

E. Song and K. Lee proposed a webservices implementation of the functionalities found in the IEEE 1451.0 and IEEE 1451.5 through a Simple Object Access Protocol. In their test implementation, a Client could connect to an STWS compatible NCAP and read its corresponding Transducer Electronic Data Sheets [22]. D. Wobschall implemented a Network Capable Application Processor based on serial communication for communicating to sensors and ethernet for internet access. While this work was done in 2002, the main principles of communication and the structure of the related TEDS are still carried out through the standard today [23]. A. Fatecha, J. Guevara, and E. Vargas proposed a reconfigurable architecture for the smart sensors within an IEEE 1451 network. This method would utilize a Programmable System on a Chip to adapt the system to multiple types of transducers, while utilizing TEDS to manage the configuration [24]. Ma et al. developed a 1451-2 and -4 compliant data acquisition module. They combined the mixed-mode interface defined in 1451-4 to access TEDS with the serial interface defined to communicate with an NCAP [25].

W. Kim et al. attempted to integrate the IEEE 1451 family of standards and the Health Level 7 standard for exchanging sensor data from multiple medical devices. The main approach taken by the researchers focused on the presentation of data, formatting the HL7 data to better meld with the TEDS structure within IEEE 1451. The communication of devices was another aspect which the research used IEEE 1451 to solve by addressing devices according to the standard [26]. A more traditional application of TEDS was implemented by Croitoru et al. who developed TEDS in an I2C based EEPROM connected to a PIC32 microcontroller. The main contribution was showing that TEDS could be located outside the sensor or transducer which allows for more flexibility in what transducers could be used in a 1451 design [27]. Another team focused on communicating TEDS information through a plug-and-play interface, Hernández-Rojas et al., presented a framework for virtualizing this information. Instead of storing the information physically on memory on the sensors or on-board the system, the TEDS information could be stored in any entity within the network, or in a cloud service. Traditionally in the standard, to reconfigure any TEDS information, you needed to physically access the memory with a programmer or design the sensor node with this ability. With a virtualized TEDS, information can be more easily accessed and modified as needed, also allowing for easier retrofitting of older or existing systems [28].

Chapter 2

An Overview of IEEE 1451

The IEEE 1451 family of standards can be more easily understood by looking at what entities are required in an IoT network to facilitate communication between a transducer and an end-user. A diagram of how this interconnection is found in Figure 1.

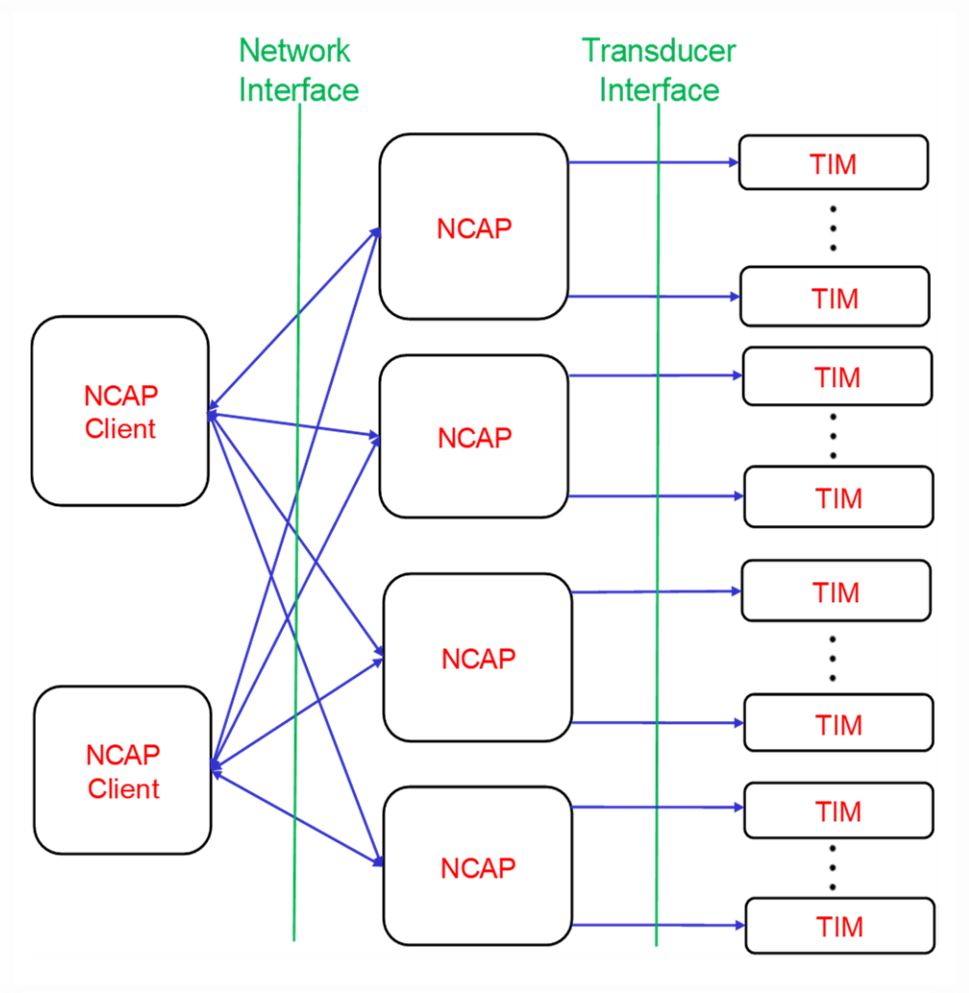


Figure 1. Communication flow diagram for P21451 based transducer networks.

NCAP Client

At the highest level, there is the NCAP Client Abstraction (referred to as the NCAP Client) which interfaces with the entity which wants to control an actuator or read some sensor value. The NCAP Client could be a variety of users or technology. In the case of a Smart City, or an extremely distributed system comprised of many subnetworks, each one of these sub-networks could act as an NCAP Client for another one of the networks.

An example of this can be conceptualized when thinking about a city-wide network which utilizes both personal vehicles and public transport to monitor conditions within a city, much like the networks talked about in [13]. A municipality could use this type of network to monitor traffic flow and better the performance of their public transport systems by dynamically routing buses to areas with less congestion. If these systems were kept as two separate networks (one interfacing with the personal vehicles and one interfacing with the bus system), whenever the bus network wishes to check on the traffic flow in the car network, the bus network then becomes an NCAP Client to the personal car network. The main server in charge of planning routes for the busses may wish to probe specific sectors of the city given time of day or knowledge of a traffic incident within the city. If a new path is found or if major delays are imminent, bus drivers can be notified and they can change their routes accordingly.

As for the everyday person using public transport, if the city were to team up with a service like Google, a smartphone application could be made available, which allows a person to not only see real-time traffic flow but also have access to the real-time scheduling of the bus system. In this case, an application user is an NCAP Client, since they are requesting the information. However, in most cases, what the application user is seeing is either a modified webpage or receiving public information from a server. The server would be the one requesting information in a timed manner and would also be the part of the network to notify all users of the application if something were to happen in the public transit system. This all depends on how much information and functionality the designer of the application and the network wants to make available to the public. In this case, both the end-user and the server can be lumped together into a single NCAP Client abstraction layer. Applications like this already exist in major cities such as New York City and London, however they focus more on the mass transit systems such as the subways and buses.

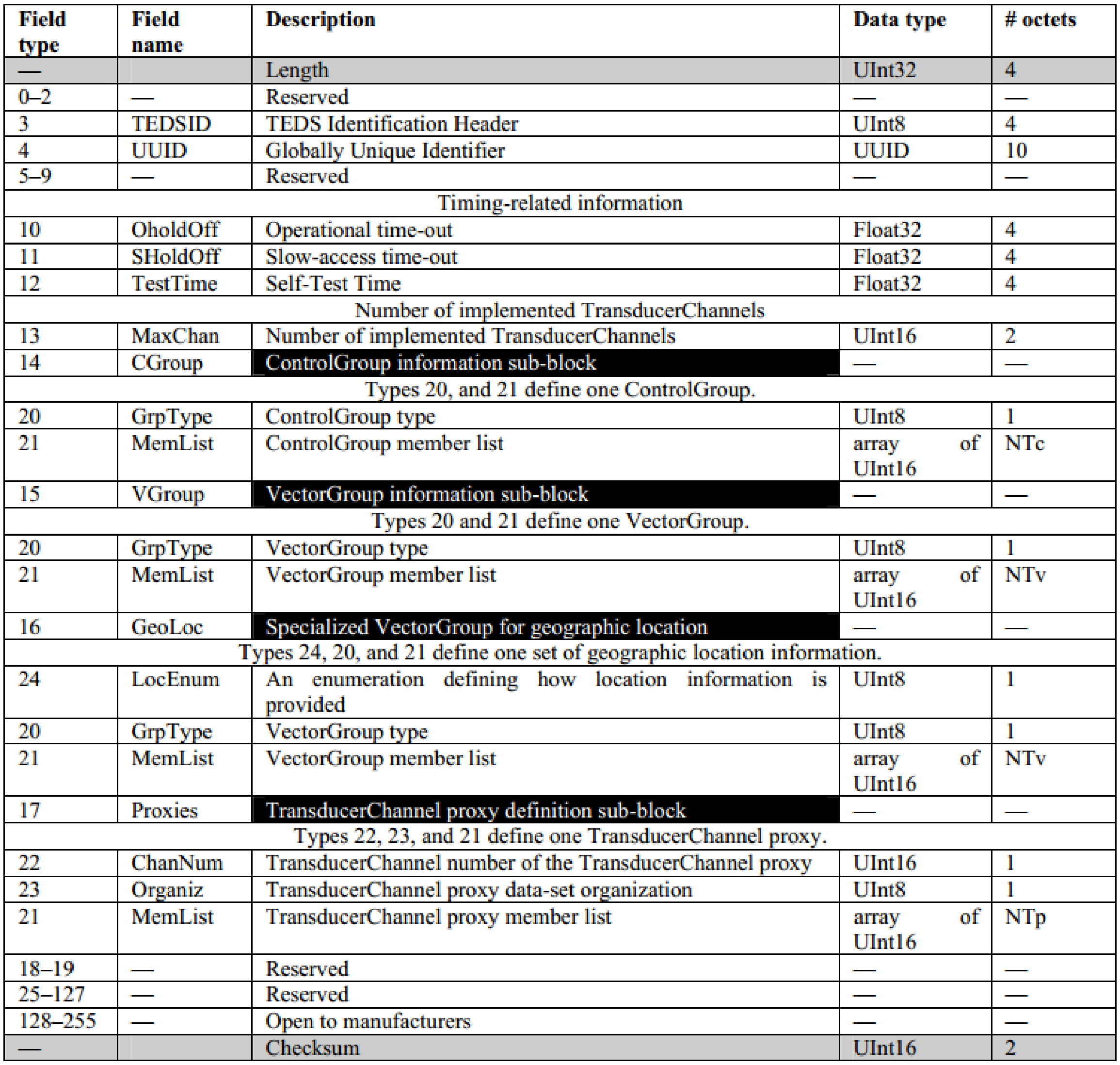
NCAP Server

Using an abstraction-based approach, the rest of the sensing network does not need to know exactly how the NCAP Client is implemented, only whether it can communicate with it and what functions it needs available to it. The NCAP Client is assumed to be talking to the network over the internet, whether that is via WiFi, cellular data, etc. The NCAP Clients, however, do not necessarily have a point-to-point communication channel with the transducers themselves. What the Client is communicating with is one of potentially multiple Network Capable Application Processor (NCAP) servers, which act as the brokers between the open internet and the actual sensing network. In systems such as the smart city example there is public-facing data and private data which can make the task of point-to-point communication more complex.

Transducer Interface Module (TIM)

Once a request for some information is reached at the NCAP Server level, it is then mapped to its corresponding function call as well as which method of communication it will use to talk to a specific Transducer Interface Module (TIM). TIMs contain the necessary circuitry to power and condition any transducers attached to it as well as contain the processing power to communicate with an NCAP Server. This communication is not just limited to one medium, with implementations utilizing techniques such as ZigBee, BlueTooth, Inter-Integrated Circuit (I^2C) serial communications, and more. Each one of these methods of communication is covered within the standard, with several having their own set of Transducer Electronic Data Sheets (TEDS) which are stored in memory either on the transducers themselves or within the memory of the TIM. Along with communication specific information such as baud rate, TEDS contain information about the TIM as a whole as well as each transducer connected to it, ranging in content from what company manufactured it to calibration coefficients. Every TIM is required to have a set of MetaTEDS, which contain the information as seen in Figure 2.

These TEDS facilitate one of the largest benefits in IEEE 1451 enabled networks, the ability to “hot swap” TIMs and even transducers within the network. Hot Swapping is the process of adding or taking away parts of the transducer network without requiring the system to fully shutdown or restart. A similar example of this can be seen when using a flash drive with a computer. When the user plugs the flash drive in, the operating system recognizes the new drive and begins to ready the system to communicate with the new device. Once the user is finished with using the drive, the computer can ensure there is no more communication between it and the flash drive, and then the user can remove it. During this entire process, the computer never had to shut down or reboot in order to begin or stop using that drive. This same idea exists within this standard, where the TIM takes the place of the flash drive and the NCAP Server is the computer.



Field type	Field name	Description	Data type	# octets
—		Length	UInt32	4
0–2	—	Reserved	—	—
3	TEDSID	TEDS Identification Header	UInt8	4
4	UUID	Globally Unique Identifier	UUID	10
5–9	—	Reserved	—	—
		Timing-related information		
10	OholdOff	Operational time-out	Float32	4
11	SHoldOff	Slow-access time-out	Float32	4
12	TestTime	Self-Test Time	Float32	4
		Number of implemented TransducerChannels		
13	MaxChan	Number of implemented TransducerChannels	UInt16	2
14	CGroup	ControlGroup information sub-block	—	—
		Types 20, and 21 define one ControlGroup.		
20	GrpType	ControlGroup type	UInt8	1
21	MemList	ControlGroup member list	array of UInt16	NTc
15	VGroup	VectorGroup information sub-block	—	—
		Types 20 and 21 define one VectorGroup.		
20	GrpType	VectorGroup type	UInt8	1
21	MemList	VectorGroup member list	array of UInt16	NTv
16	GeoLoc	Specialized VectorGroup for geographic location	—	—
		Types 24, 20, and 21 define one set of geographic location information.		
24	LocEnum	An enumeration defining how location information is provided	UInt8	1
20	GrpType	VectorGroup type	UInt8	1
21	MemList	VectorGroup member list	array of UInt16	NTv
17	Proxies	TransducerChannel proxy definition sub-block	—	—
		Types 22, 23, and 21 define one TransducerChannel proxy.		
22	ChanNum	TransducerChannel number of the TransducerChannel proxy	UInt16	1
23	Organiz	TransducerChannel proxy data-set organization	UInt8	1
21	MemList	TransducerChannel proxy member list	array of UInt16	NTp
18–19	—	Reserved	—	—
25–127	—	Reserved	—	—
128–255	—	Open to manufacturers	—	—
—		Checksum	UInt16	2

Figure 2. MetaTEDS Content

As mentioned before, each TIM can contain multiple transducers where each transducer has its own Transducer Channel. This allows for easier access to specific sensors and allows each channel to have its own ChannelTEDS describing what the transducer is. In some cases, a transducer may have multiple channels assigned to it. This is normally seen with complex transducers which have multiple moving parts or readings. An example is the DHT11 temperature and humidity sensor, which returns both humidity and temperature information to the TIM. To demonstrate the use of multiple channels for one transducer, temperature and humidity readings are separated into two channels. When the client requests temperature, a request is formed for Channel 1 of the correct TIM; for humidity, the request is for Channel 2. This allows complex systems, such as a variable frequency drive, to be managed by a single TIM.

Network Architecture

These parts of the network come together to form an IEEE 1451 Smart Transducer Network as seen in Figure 3. In this figure, there is a single NCAP Client, a single NCAP Server, and a single TIM with multiple channels. This is a block diagram of one of the early implementations created to begin discovering more about what is necessary to create a basic smart transducer network with this standard. The initial idea was to simulate a smart home, utilizing a simple Android application to control both a light and a fan while being able to request the current temperature in the room. Using this basic implementation, different internet communication protocols, as well as multiple hardware implementations, were tested for the best suited for an application at Rowan.

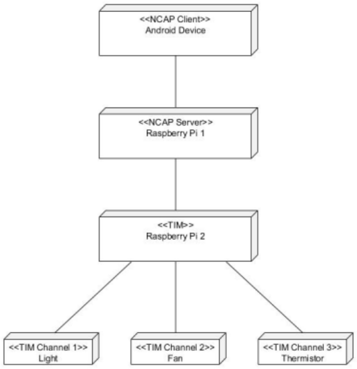


Figure 3. Block Diagram of a IEEE 1451 network consisting of one NCAP and

Defining the NCAP Server

It is the job of the NCAP Server to act as the liaison between the open internet and a closed network of transducers. Because of this, the services required to act as this gateway vary greatly, as seen in Figure 4. A communication stack must be managed to be able to utilize protocols such as XMPP, UDP, etc. The NCAP Server also needs to keep track of the NCAP Clients and TIMs registered to it, as well as manage when these entities want to enter or leave the network. This communication stack also manages the proper drivers and stacks to communicate with different TIMs, and then compose the correct style message to communicate with those TIMs.

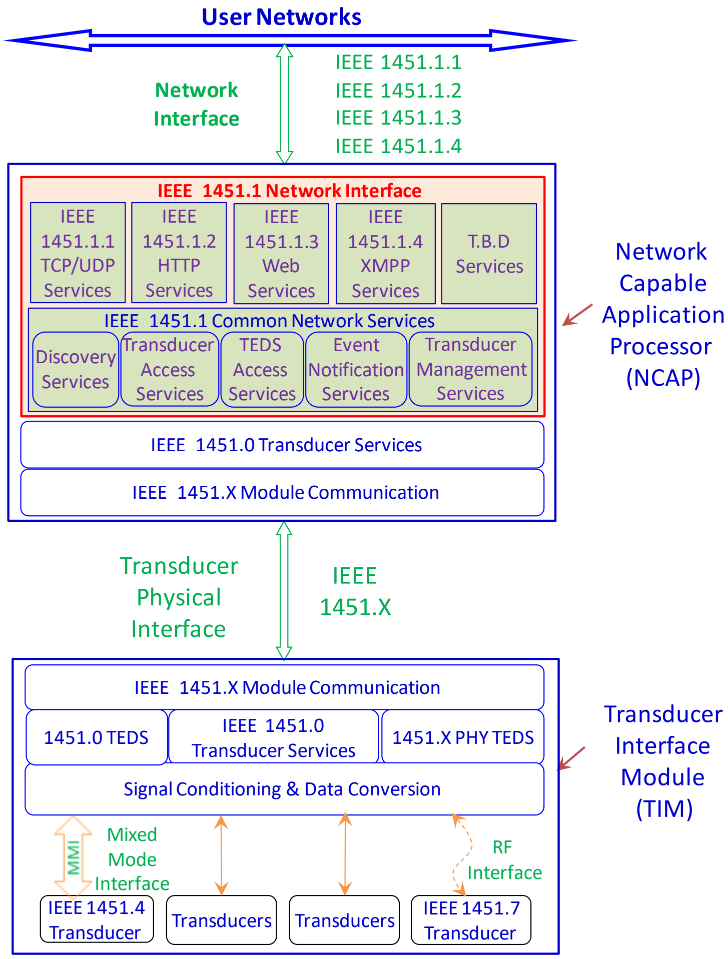

User Networks
Network Interface
IEEE 1451.1.1
IEEE 1451.1.2
IEEE 1451.1.3
IEEE 1451.1.4
IEEE 1451.1 Network Interface
IEEE 1451.1.1 TCP/UDP Services
IEEE 1451.1.2 HTTP Services
IEEE 1451.1.3 Web Services
IEEE 1451.1.4 XMPP Services
T.B.D Services
IEEE 1451.1 Common Network Services
Discovery Services
Transducer Access Services
TEDS Access Services
Event Notification Services
Transducer Management Services
IEEE 1451.0 Transducer Services
IEEE 1451.X Module Communication
Network Capable Application Processor (NCAP)
Transducer Physical Interface
IEEE 1451.X
IEEE 1451.X Module Communication
1451.0 TEDS
IEEE 1451.0 Transducer Services
1451.X PHY TEDS
Signal Conditioning & Data Conversion
MMI
Mixed Mode Interface
RF Interface
IEEE 1451.4 Transducer
Transducers
Transducers
IEEE 1451.7 Transducer
Transducer Interface Module (TIM)

Figure 4. IEEE 1451 Reference Model

Communication Models. In the IEEE 1451-1 current draft, there are defined three different communication models which govern the flow of information and responsibilities of each part of the NCAP, or how the information is exchanged from the Client to the NCAP.

Client-Server model. The most fundamental communication model within internet communications is the Client-Server communication model. It is defined as a relationship between two computer programs in which one program, the “Client”, makes a service request to another program, the “Server”. The Server then provides its services to fulfill the request. The Client and the Server both typically communicate over a computer network on separate hardware; however, this does not mean that they both cannot reside in the same system. This model supports communication modes such as synchronous, point-to-point, and one-to-many. As can be seen in Figure 5, the NCAP Server (Server) provides a service based on the request from the NCAP Client (Client). These two terms are encapsulated under the NCAP since it is the entity which is responsible for communicating over the internet/network.

While common in other applications, this model does not provide sufficient services required to sustain an IoT enabled transducer network. An example of how this model is not robust enough would be a smart home with a security system. If the security system relied on the homeowner constantly sending requests to the NCAP Server to check if any alarms are triggered, there is a risk that the notification of an event could be delayed or even missed. To allow for a close to real-time response from the system, the NCAP Server needs to asynchronously send a message to an NCAP client.

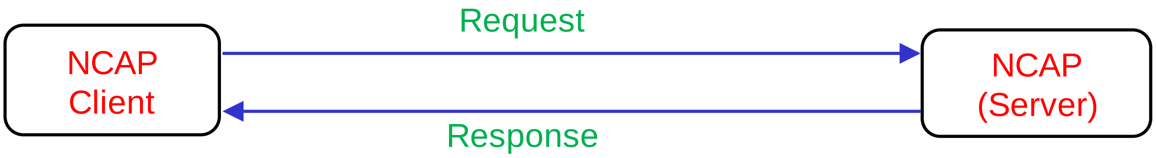


Figure 5. Client-Server Communication Model

Publish-Subscribe Communication Model. In the publish-subscribe model, instead of focusing particularly on the physical entities, the linkages between sources of data and consumers of the data are defined. An NCAP Server would be able to provide data for consumption by the NCAP Client. The Publish-Subscribe model is more loosely coupled than the client-server model and provides asynchronous communication. Applications and services that publish information typically do so without waiting. As seen in Figure 6, this model also allows for multicast publication or a many-to-many form of communication. This brings with it the functionality to communicate to multiple NCAP Clients who are subscribed to a particular "topic" of information. The most common topic, which is utilized in a smart transducer system, and possibly one of the most vital, is an Event.

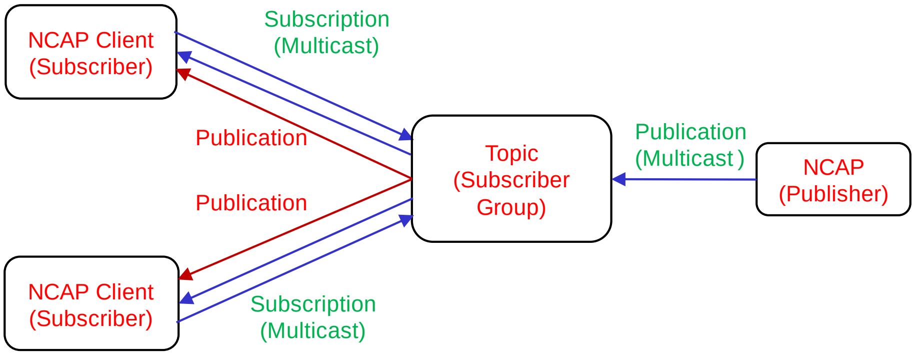


Figure 6. Publish-Subscribe Model

Event Notification Communication Model. The noticing and coping with events are vital for many systems, and event management could be one of the leading drives towards making transducer networks smarter. In the case with measured values, notifying a controller or user about certain thresholds being surpassed is vital for maintaining the health of the system and its operations. For example, if a factory producing raw fish products is not notified of temperatures in their holding tanks or freezers, potential health issues can arise. Events can also be about the status and health of the network, from the arrival of a new sensor or transducer to noticing that the connection between the NCAP and the network has a high packet loss rate. Due to the number of different types of events that can be present in any given system and the priority these types of message take, a third communication model which handles these events needs to be defined.

As seen in Figure 7, the Event Notification model is almost the exact same layout as the previous publish-subscribe model. The only difference coming with the addition of the “event” tag as part of the topic. This is to add priority to the message sent from the NCAP Server to the NCAP Client. This “event” tag also highlights the need for message validation (whether the message was received), self-automation (performing some immediate task with the need for NCAP Client permissions), and other considerations.

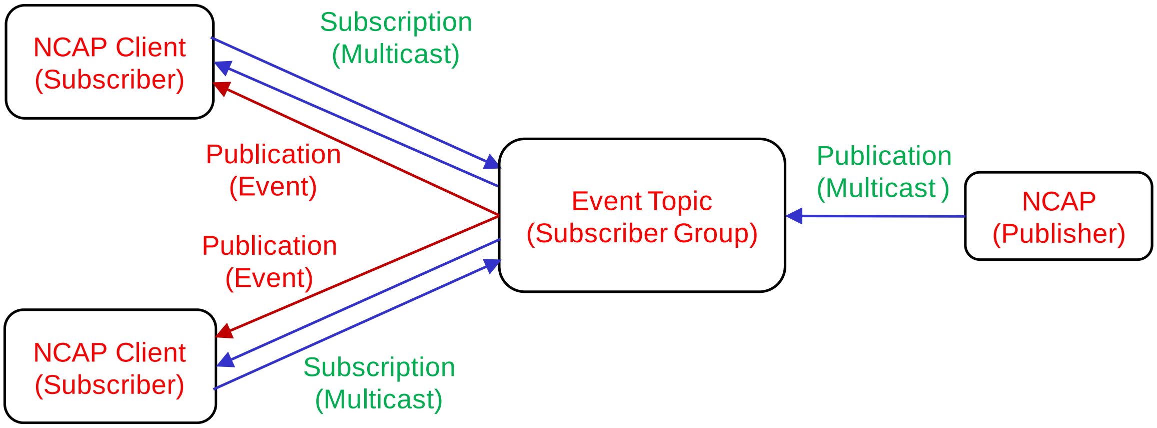


Figure 7. Event Notification Model

NCAP Functionalities

With these communication models in mind, it is now the task of the standard to lay out the services and functions which the NCAP Server can provide and what the NCAP Client can request. These functionalities are divided into five main service types: Identification, Transducer Data Access, TEDS Access, Event Notification, and Transducer Management. For each group of services, a specific communication model will be referenced to show whether these services are synchronous or asynchronous, as well as if they may require sending messages to multiple entities.

Identification Services. Identification services are utilized whenever an entity joins or leaves the entire network, as well as used to track changes in the network. These services are broken down into three main functional groups: Registration, Discovery, and Joining. Registration mainly deals with the NCAP Server or a TIM turning on or off and registering itself with the network. For example, when the NCAP Server is initially powered on, it begins to broadcast over the internet that it is available to be connected to, as can be seen in Figure 8. The request is sent every second and is treated as a broadcast to any NCAP Client that is willing to listen.

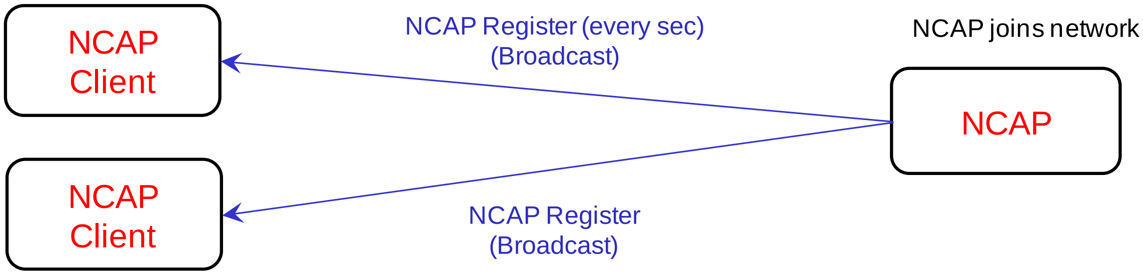


Figure 8. NCAP Server Register Service

In the same way, if the NCAP Server needs to shut down (possibly due to situations like low battery), the NCAP Server will send an NCAP Unregister Request as seen in Figure 9 so that the NCAP Client can handle the departure. It should be noted that the specifics in the content of these messages and any headers rely on other members of the standard family, such as IEEE 1451-1-4.

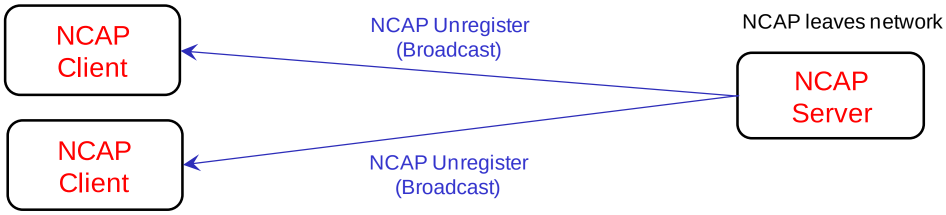


Figure 9. NCAP Unregister Service

Another part of the registration portion of the Identification Services for the NCAP Server is to manage the registration of TIMs which are connected to it, as well as the transducers which are connected to those TIMs. One major feature that separates the registration portion of the Identification Services and the rest is that these services are NCAP Server initiated. Instead of waiting for a request from an NCAP Client, the NCAP performs these services on a schedule.

Discovery Services are the services which the NCAP Client have access to and can initiate with a request to the NCAP Server. These provide an NCAP Client the ability to discover what is connected to the network from the NCAP Server level down to the individual transducers. Unlike the registration portion of the services, the discovery services are based on the "Client-Server" communication model, wherein the NCAP Client needs to send a request for information from the NCAP Server. At the highest level, there is the NCAP Server Discover service in which the NCAP Client receives a list of all the NCAP Servers available on the network. As can be seen in Figure 10, the NCAP Client sends the request out on the entire network and any available NCAP Servers return a message containing identifying information.

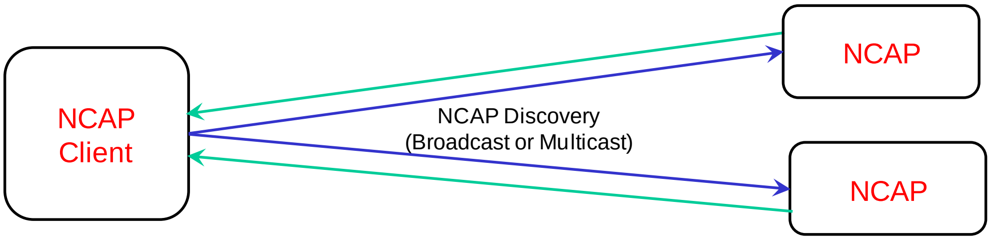


Figure 10. NCAP Discover Service

The NCAP Client can also query a particular NCAP Server as to what TIMs are connected to it. This TIM Discovery Service, as seen in Figure 11, is initiated by the NCAP Client and targets a specific NCAP Server. Upon receiving a TIM Discovery Service request, the NCAP server then proceeds to locate all TIMs connected to it either physically or wirelessly. The NCAP Server will then compile the TIM Identifications, TIM Descriptions, as well as the number of TIMs connected and send a response back to the NCAP Client.

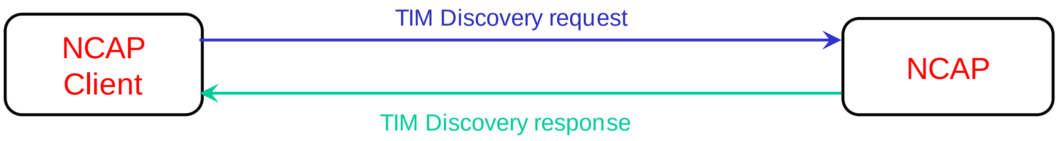


Figure 11. TIM Discover Service

An NCAP Client can also request information about the Transducers attached to a specific TIM by sending a Transducer Discovery Service request. As seen in Figure 12, from the viewpoint of the NCAP Client, the communication flow is the same as the TIM Discovery Service; however, there is more information both sent and received during this message. An NCAP Client sending this request tells the target NCAP Server which TIM it wishes to know more about. After performing the proper function calls as will be seen in later sections, the NCAP Server will compose a message containing the number of transducer channels as well as the Channel IDs of each one. This ChannelID does not contain information about what the transducer is. The request for this information is handled by the TEDS Access services; however, this service provides the proper identification information for these transducers.

To make a grouping of NCAP Servers possible, the NCAP Client Join and Unjoin services have been added to the current draft of the standard. These services are initiated by the NCAP Client much like those functions seen in Figure 11 and Figure 12, but have the same contextual information as the Registration Services. An NCAP Client can send an NCAP Client Join request to an NCAP, which forms a group between those two entities. An NCAP Client can also leave a group by sending an NCAP Client Unjoin request. An NCAP Client can also request a current roster of the participating NCAP Servers and NCAP Clients within a group. These groups enable the use of the Publish-Subscribe Model, wherein a message being sent by an NCAP Server can be sent to NCAP Clients as well as other NCAP Servers.

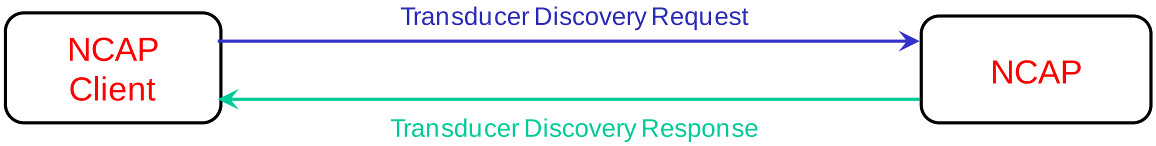


Figure 12. Transducer Discovery Service

The addition of these services allows for much easier implementations of background processes such as health reporting of a TIM or NCAP Server as well as open the door to semi-autonomous smart transducer networks. In a factory setting with high-temperature oven or processes, there may be hard and soft thresholds in place. The soft thresholds are ones which could damage the product being heated or alter a material property undesirably in a finished product. A hard threshold could be considered a point of no return, where the temperature exceeds operating thresholds. For each of these thresholds, different procedures may be used to force the system back to a normal operating range. If the temperature rises above the soft threshold for a given process, a normal event notification message sent to a plant controller may be enough to handle the situation. If the process goes over a hard threshold the NCAP Server, which is communicating to the monitoring TIM, could send out an event notification. However, instead of just going to an NCAP Client, this message is also sent to another NCAP Server within a group. The notified NCAP Server could then proceed to initiate emergency procedures while the plant controller can halt operations in other parts of the factory safely.

Transducer Data Access Services. The Transducer Data Access Services allow for the reading or writing to specific transducers within the network. The data varies between a single point for/from a single transducer to data for multiple transducers at a time. To facilitate easier access to timed sequences of data, the concept of Block Data is introduced. Unlike the previous Registration Services, the NCAP Client is required to send more information in its initial request as to identify within the network a transducer. This information includes an NCAPID, TIMID, and a Transducer Channel ID to isolate one transducer. Other runtime parameters such as timeouts (how long should the NCAP Server attempt to retrieve the information before giving up), how many samples are required, which sampling time should the data be taken, and more. Due to the requirement of having this information, these services are normally invoked after initializing the network and after the NCAP Client has discovered all the entities it requires.

The basic division among these services depends on whether the requested information needs to be read or written. Read services deal with acquiring sensor data; write services set the transducer to a value. By definition, a transducer can contain both sensing and actuating elements; for example, a servo motor can have its position set as well as have its current position read. When calling the services, however, the information fields which need to be filled out are essentially the same for reading and writing. From here, the Transducer Access Services can be grouped into three main categories: Synchronous, Asynchronous, and Secure.

Synchronous Services. Synchronous communications are built off the Client-Server Communication model, where both communicating parties must be present during the entire exchange of messages. Doing this allows both communicators to be time synchronized. As can be seen in Figure 13, there is a message pair which is sent by both parties during this service. This request-response sequence is common among all the Synchronous services, and each service has its own specific set of information required to successfully complete the request.

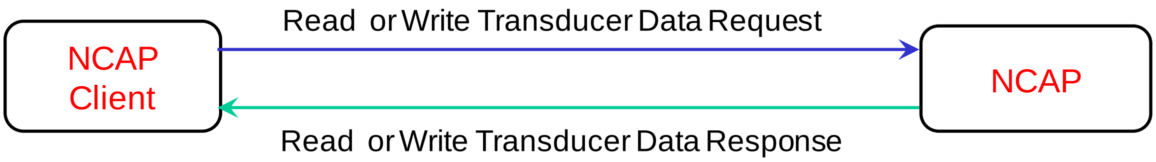

Read or Write Transducer Data Request
NCAP Client
NCAP
Read or Write Transducer Data Response

Figure 13: Synchronous access of transducer data

One of the two fundamental building blocks required for these services to work is the “Read Transducer Sample Data From A Channel Of A TIM” service. This is a request for the current state or measurement of a transducer connected to a specific Channel on a specific TIM on connected to a specific NCAP Server. The request sent by the NCAP Client must contain an NCAPID, TIMID, ChannelID, Timeout, and Sampling Mode on order for the NCAP server to query the correct TIM in the specified fashion. After the NCAP Server retrieves the information from the TIM (with the correct TIMID), the NCAP Server forms a response containing an ErrorCode, NCAPID, TIMID, ChannelID, and finally the TransducerSampleData.

An example request for a single point of data would be structured as: (7211 [FunctionID], 1 [NCAPID], 1 [TIMID], 1 [ChannelID], 10 [Timeout in Seconds], 5 [Immediate Sampling Mode]). The response from this (assuming there were no errors during execution) would look like: (7211 [Function ID], 0 [Error Code], 1 [NCAPID], 1 [TIMID], 1 [ChannelID], 243 [Transducer Data]).

At first glance there seems to be redundant data being sent back and forth in this exchange; however, this “redundancy” can help solve some issues which may arise in these systems. The first is security. This type of request-response allows for verification that the response the NCAP Client is receiving is genuine, making it where basic attacks on the system such as flooding the communication channel with random information can be negated. The second reason is the need for an NCAP Client to manage multiple conversations at the same time. While waiting to hear about the reading of a transducer, a new NCAP Server may come online. Without the information such as the Function ID and the NCAP ID, it could be very easy for the NCAP Client to confuse this “NCAP

Server Arrival” message with the response from the read request. This situation may seem very situational; however, in a complex IoT network, there are going to be a lot of messages sent to an NCAP Client. By having the information replicated in the response, the NCAP Client can then manage these messages and link the response to the proper request.

The message structure of the single channel transducer read service is common among the other synchronous read functions; however, each service adds its own extra field or two to help set up the acquisition session. For example, the “Read Transducer Block Data From A Channel Of A TIM” service has the same fields as the previous service; however, the Request also has the addition of the NumberOfSamples, the SampleInterval, and the StartTime, while omitting the SamplingMode. This is because block data is a time-series based measurement with equally spaced samples of data, and as such, the systems needs to know when to start measuring, how much time needs to be in between each sample, and how many samples are being requested. The response looks extremely similar with the only difference being the data presentation. The request would look like (FunctionID [7212 for block read from single transducer], NCAPID, TIMID, ChannelID, Timeout, NumberOfSamples, SampleInterval, StartTime). The response would be (FunctionID, ErrorCode, NCAPID, TIMID, ChannelID, TransducerBlockData).

The ability to read multiple points of data at a given time needs to also be implemented. For example, a control server may want to query a TIM inside a room within a house for the temperature, light level, and humidity. Or the server may need to check the temperature of multiple rooms, requiring the request to be sent to multiple TIM’s that are attached to the same NCAP Server. Using what has been defined so far, three different messages would need to be sent, sorted, and managed for three different responses. Each one of these messages would have almost the exact same content, except for the different transducer channels. It is for this reason that there are resources built into these services to request from multiple transducers at the same time.

To allow for this in services such as "ReadTransducerSampleDataFromMultipleChannelsOfATIM" and "ReadTransducerBlockDataFromMultipleChannelsOfATIM", only one field must be changed in the request and responses of these functions. Instead of declaring a single ChannelID, an array of Channel IDs corresponding to the transducers where data is to be retrieved. In the case where different channels are on the same TIM, the request would look like: (7213 [FunctionID for Single Data from Multiple Channels], NCAPID, TIMID, ChannelIDs [ex. {1, 3, 4}], Timeout, SamplingMode). The multi-channel block read request is also formatted the same as the single channel counterpart, only with an array of ChannelIDs instead of just one.

This same approach is taken when an NCAP Client wants to access transducers from multiple TIMs. Out of all the Synchronous services, the read requests for "Multiple Channels Of Multiple TIMs" have the most complex syntax due to the amount of information needed. Instead of having a single TIMID, there needs to be an array of accessible TIMs. Since the NCAP Client may not want information about the transducers on the same set of ChannelIDs, there needs to be an array of ChannelIDs for **each** of the TIMIDs in the request. This could be difficult for a program to figure out exactly which ChannelIDs go with each TIMID. To aid in this, a new field is added in the request which states how many Channels of each TIM need to be read. With all of this in place, the

"Request for Reading Transducer Sample Data From Multiple Channels Of Multiple TIMs" would look like: (7215 [FunctionID], {1,2} [TIMIDs], {3, 1} [NumberOfChannels], {{1, 3, 4}, {2}} [ChannelIDs], 10s [Timeout], 5 [SamplingMode]). The NCAP Server sends a similar response back to the NCAP Client once the acquisition is completed, containing arrays of data for each of the TIMs.

Asynchronous Services. The functions within the Transducer Access Services that have been presented so far have been synchronous, where the NCAP Client waits for a specific response from the targeted NCAP Server. This model of communication is not applicable to all situations, such as receiving data from a transducer over a long period of time. For example, a system may require the measurement of a specific transducer of the period of a day at a sampling rate of 1 measurement per minute. Using the Synchronous functions provided in the standard, the NCAP Client could expect a message containing 1440 data points, which could be difficult to send through a communication channel all in one packet. This also can be difficult to parse through and manage efficiently in program memory on both the NCAP Server and Client. There could also be systems where an NCAP Client wished to monitor an entire subsystem in a process, which may contain large amounts of transducers over a long period of time (such as a day). It is for these reasons that an Asynchronous Communication Model is needed to facilitate long term or "indefinite" term acquisition requests.

The communication model as seen in Figure 14 still requires the "handshake" request-response messages that are present in Synchronous services; however, there is a callback message that is only sent from the NCAP Server to the NCAP Client. The flow of communication begins with the NCAP Client initiating the acquisition session, sending a request message containing similar information as the Synchronous Block Read service. The NCAP Server will then ready itself for the acquisition and once ready, will send a Response back to the NCAP Client containing an ErrorCode corresponding to any issues it may have had, as well as an Operation Identifier (OperationID).

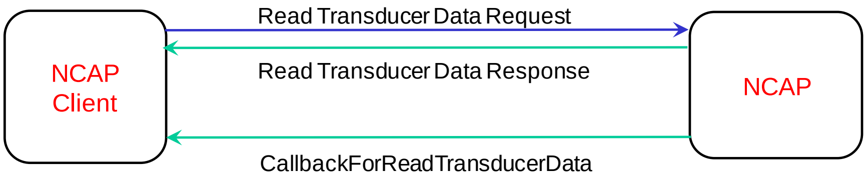

Read Transducer Data Request
Read Transducer Data Response
NCAP Client
NCAP
CallbackForReadTransducerData

Figure 14. Asynchronous access of transducer data.

After the Response is sent, and if no error occurred during the initialization, the NCAP Server begins to acquire data. Because this operation could take tens of seconds to hours, the NCAP Server will compile the data into one message and send it once the acquisition is completed. This frees up the NCAP Client to perform other functions while waiting for the Callback from the NCAP Server. The Callback contains roughly the same information as a BlockData response from the synchronous services. Each Asynchronous process which is currently running on the NCAP Server is assigned an OperationID once the initial request is received from the NCAP Client. This allows for multiple processes to run on the same NCAP Server at any given instance. The OperationID also allows the NCAP Client to map any Callback received to a corresponding open function call.

Because these services are meant to be used over periods of time, the data sent to the Client will be in the form of blocks. In the same fashion as their Synchronous counterparts, the ReadBlockData services for single Channel, MultiChannel, and Multi TIM requests contain the ID's required to isolate a particular transducer within the network, sampling intervals, duration, and a timeout. The only function unique to the asynchronous transducer access services is the ReadTransducerStreamData service. This service is very similar to those dealing with BlockData, however, instead of requesting a specific number of data points separated by a sampling interval, start and end times are used. A sampling interval is still utilized to acquire data; however, this service makes it easier to schedule services that run at specific times.

TEDS Access Services. Every TIM in the network contains TEDS. Depending on the way in which the TIM communicates to the NCAP Server, there are a variety of different TEDS which may need to be read. Some sections of different TEDS can be edited by the NCAP Client. Both the reading and writing services which are available for use follow the Request-Response communication model as can be seen in Figure 15. For the sake of the theory of operation, the read and write versions of the services have a similar form to Requests and Responses.

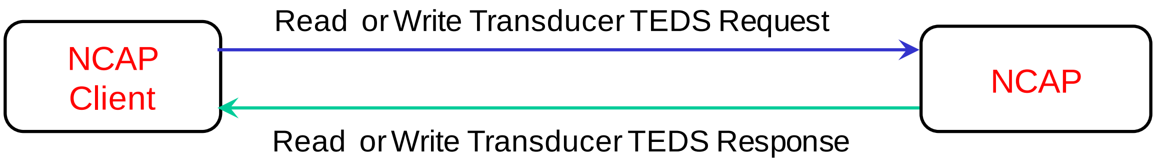

Read or Write Transducer TEDS Request
NCAP Client
NCAP
Read or Write Transducer TEDS Response

Figure 15. TEDS Access Services Communication Model

For the Read TEDS Services, the NCAP Client initiated Request contains the FunctionID, NCAP_ID, TIM_ID, and a Timeout duration. The only exception to these included pieces of information are the Read Services dealing with specific Transducer Channels within a TIM. In these cases, the addition of a ChannelID is required to ensure the NCAP Client is targeting the correct Transducer. As for the Responses, the Read TEDS services respond only with an ErrorCode and an array containing the requested information. Write TEDS Services are very similar in style to the Read TEDS Services, with each one requiring the same information to perform the task. All Write TEDS Services Requests contain a FunctionID, NCAP_ID, TIM_ID, Timeout, and an array containing the information to be written. As with the Read Services, those services which target specific Transducers require a ChannelID in addition to the other information. The Response from the NCAP Server is simply an ErrorCode.

Although the TEDS Access Services seem very familiar, the distinguishing piece of information which allows both the NCAP Client and NCAP Server to access the specific set of TEDS is the FunctionID. Every type of accessible TEDS has its own unique FunctionID within the standard, meaning that a WriteTIMMetaIdTEDSservice Request (FunctionID 7319) and a WriteTransducerChannelMetaIdTEDS Request (FunctionID 7320) can be differentiated and handled accordingly. Since these services require that the NCAP Client have knowledge of the NCAP_ID’s and their associated TIM_ID’s and Transducer ChannelIDs, these services are meant to be requested after the NCAP Client initializes its connection to the network by performing utilizing the Identification Services as discussed previously.

Event Notification Services. Up to this point, services related to initialization, configuration, or access to the network have been discussed. With only these services, an effective transducer network can be implemented; however, there is a lot of responsibility and effort required by the NCAP Client to ensure that the network is running correctly and to monitor for any changes within the network. In the case of an owner of a smart HVAC system in their house, this would mean that the owner's phone or device would constantly be sending out requests for transducer data, TIM and NCAP Server discovery requests, and many more. This could put quite a strain on the NCAP Client's resources, and if this is a device to be used for other purposes, it could affect the overall performance of the device. This also means there would be a lot of messages being sent between the NCAP entities, limiting the application to those which have high enough bandwidth and internet access.

When a closer look is taken, the NCAP Client should not have to constantly request a TIM Discovery service to see if any new TIMs have been connected or existing TIMs removed? What about monitoring for abnormal conditions within the sensing network, such as temperatures of transducer reaching a specific threshold? If it is assumed that during normal operation of the network, there are no new TIMs being added or removed and that the transducers will remain within their safe region of operation, then these monitoring messages are just wasted time, processing power, bandwidth, and effort. It would be more power and resource-efficient for the NCAP Client to move these responsibilities to the NCAP Server. Messages would only be sent when changes were necessary.

The Client-Server, Request-Response synchronous communication models cannot easily shift this monitoring from the NCAP Client to the NCAP Server. Instead, the Publisher-Subscriber communication model as seen in Figure 6 is utilized. This model contains two steps, Subscription and Notification, as seen in Figure 16.

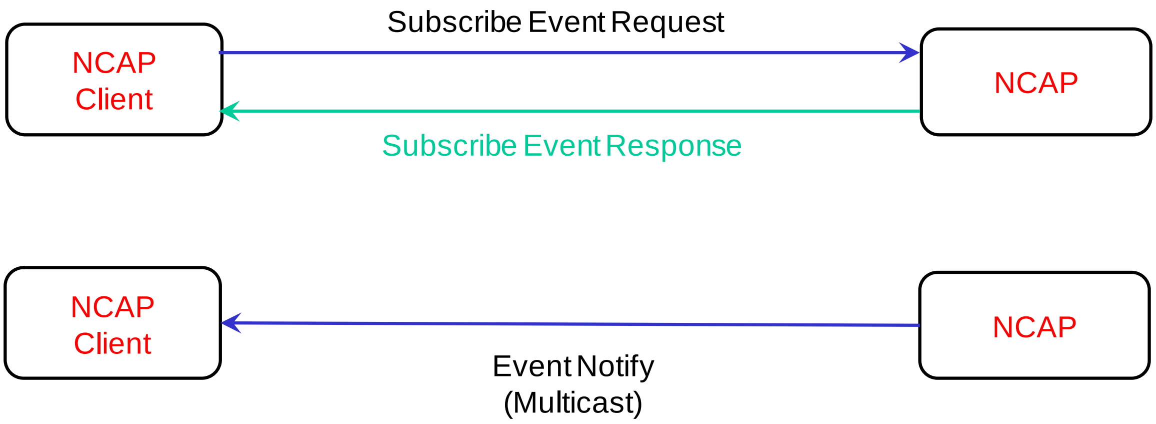


Figure 16. Event Notification Services communication model

An analogy for how this works can be seen when creating an account for a website or service. As part of the registration process, a user can choose to “Subscribe” or opt-in to receive updates from the company about different topics. The user is sending a request to the web service to be added into an email list, which the company will access when a newsletter or notification needs to be sent. Normally, the company will then send a verification email Response to ensure that the user was added to the list. From this point on, whenever the company wants to send notifications, they can simply access their list of active subscribers and send a notification email to the user, without the user having to manually request it.

To see why users would want to “opt-in” to these notifications and not be automatically subscribed to every event, social media provides an excellent example. Facebook allows users to manage what they are notified about from the friends and company pages they are subscribed to. Imagine if upon adding a new friend or subscribing to a new group, a user was automatically notified whenever they did so much as “like” something or made a new post. The user would be so overwhelmed in the amount of available information from these notifications, that trying to find the useful information that pertains to a specific thing or topic would be almost impossible. But this does not limit what information can be found, as a user can still manually request information about a specific person and see what has changed on their profile or wall.

There are two main events that occur within the network that could be placed under the responsibility of the NCAP Server to monitor that would greatly reduce the amount of bandwidth and messaging overhead for the NCAP Client. The first is a group dealing with the arrivals and departures of TIMs to and from the network. Whether these are caused by someone manually removing or adding the TIM, or by a malfunction in the electronics, these events are necessary to monitor since the NCAP Client could then automatically perform a set of Identification Services to register the change. The second set of notifications deal with alerts from the transducers, specifically dealing with the passing and setting of thresholds. As discussed, each service has two components: the traditional Request-Response for subscribing to an event, and then the Notification to announce when the event occurs.

TIM Arrival and Departure Notification Services. At the NCAP Server level, as a new NCAP Server comes online and tries to become part of the network, it broadcasts a message every second to catch the attention of interested NCAP Clients. NCAP Servers which decide to leave the network due to power issues or some other reason also have a service available in the Identification Services to announce their departure. These services would also be useful at the TIM level to notify the NCAP Client when new TIMs, and in turn new transducers, are available to retrieve data from. However, since a point-to-point connection between the NCAP Client and the TIM does not exist, this service must be managed through the NCAP Server. This means that the NCAP Server, not the NCAP Client, must check for changes with the TIMs connected to it.

The Subscription Request to be notified of the arrival of a new TIM must contain the proper FunctionID, NCAP_ID for which NCAP Server needs to be monitored the changes, a Request Timeout, and a “Subscriber” which is a string which the NCAP Client and Server uses to manage all active subscriptions. Upon receiving and executing the Request without error, the NCAP Server will send a Response containing the FunctionID, an ErrorCode, a NewTIMPublisher string, and a SubscriptionID much like that seen in the Asynchronous Transducer Access Services. The NewTIMPublisher is a string which both NCAP Client and Server can use to uniquely identify between different notifications sent from the same entities. This Publisher within the NCAP Server is not unique for each of the different NCAP Clients which are subscribed to it. Rather this Publisher is used in the same way as the FunctionID to determine what the eventual notification is for. This helps with managing the different Event Notification Services since there would be only one Publisher used for all the NCAP Clients rather than a unique one for each subscriber.

The NCAP Client can also use this publisher to discern between different notification sent from the same NCAP Server.

With the NCAP Client subscribed and the publication destination known to the NCAP Server, whenever a new TIM connects to an NCAP Server, it can send an AnnounceANewTIM message to the correct NCAP Clients. This message contains a FunctionID, the NCAP_ID of the NCAP Server, the NewTIMPublisher string the NCAP Server created, the SubscriptionID assigned to the NCAP Client, the TIMAnnouncementTime, the NewTIMID, and the NewTIMDescription which the NCAP Clients can use to update their roster.

Another useful notification to have is when a TIM departs from the NCAP Server. The Subscription Request for this TIM Departure Service contains the NCAP_ID, a Timeout, and a new DepartureTIMSubscriber which is used in the same way as the ArrivalTIMSubscriber. The Response to this assuming the request went through without error contains an ErrorCode, the DepartureTIMPublisher, and the SubscriptionID which is unique to each NCAP Client. Once the subscription is established, anytime a TIM is disconnected from an NCAP Server, an AnnounceADepartedTIM notification message is sent to each subscribed NCAP Client, with the body of the message containing the FunctionID, NCAP_ID, the oldTIM_ID, the SubscriptionID for the particular NCAP Client, the DepartureTIMPublisher, the AnnouncementTime, and the DepartedTIMDescription.

Transducer Alert Notification services. Transducer Alerts are primarily focused around the sensing portion of a transducer, mainly dealing with maintaining processes between thresholds. This Transducer Alert Notification can be initiated in two ways, either by the TIM directly alerting the NCAP Server or by the NCAP Server noticing a threshold has been exceeded. The TIM has the facility to alert an NCAP Server directly in the case of measurements exceeding a threshold. A TIM can only have a single threshold value (minimum and maximum) assigned to each available Transducer Channel. Some applications may require different thresholds for the same transducer, such as heat treatments or even system safety. A persistent question remains to facilitate multiple thresholds for the same transducer if the TIM can only handle one threshold. The solution for this is to allow the NCAP Server to also initiate these alerts by monitoring the readings from these transducers.

As with the Transducer Access Services, these services follow the Request-Response communication model. An NCAP Client can request changes to the thresholds of a specific Transducer Channel on a specific TIM by sending a message to the NCAP Server. This message needs to contain the following: proper FunctionID, NCAP_ID, TIM_ID, ChannelID of the Transducer, a Timeout for the Request, and the new minimum and maximum values of the threshold limits. If the Request is processed without error, the NCAP Server simply Responds with the FunctionID and an ErrorCode. The minimum and maximum range for the Transducers applies for analog Transducers, but Transducers with a finite amount of states need a little more care when setting the thresholds, as discussed in an example below. Much like the Transducer Access Services, all that is required to set the thresholds of multiple Transducer Channels at one time is swapping out the singular NCAP_ID, TIM_ID, and/or ChannelID for an array of ID's to fit whichever circumstance is needed. The threshold values in these cases need to be written as an array of threshold “pairs” with the form {MinValue, MaxValue} for each Transducer Channel.

Once the TIM level thresholds are set, the NCAP Client can also set different threshold values within the NCAP Server. These threshold alerts require the NCAP Client to Subscribe to a sensor alert, thus requiring the use of the Subscriber-Publisher communication model. The initial Request to set a threshold alert requires the FunctionID, NCAP_ID, TIM_ID, and ChannelID of the specific transducer, along with the MinMax threshold to compare against, and the Subscriber for the Sensor Alert. Upon successful processing of the Request, the NCAP Server will respond with the FunctionID, an ErrorCode, the Publisher for the Sensor Alert, and the SubscriptionID. It should be noted that a single NCAP Client can Request and set multiple Thresholds per Transducer Channel. Once any threshold in the NCAP Server’s list of active SensorAlert Subscriptions is surpassed, the NCAP Server will compile and send a SensorAlert Notification.

As previously mentioned, the NCAP Client can Request to set up alert services for multiple transducers at one time. In this case, each of the services requested is treated as single TIM, single Transducer Sensor Alert Service Requests and processed by the NCAP Server as such. This means that if **any** of the requested transducers surpass the thresholds set by that specific multi-transducer request, then a NotifySensorAlert message will be generated and sent for that sensor. It does not mean that a NotifySensorAlert request will be sent if and only if the requested thresholds are all met. More complex event structures can be developed for applications requiring them. The NCAPClient will have to manage the NotifySensorAlerts from each transducer.

There could be transducers which only have True and False as possible states and an NCAP Client would want to be alerted when the state of the Transducer is True. If the True state can be represented as a 1 and the False state as a 0, then a SetupSensorAlertThreshold request can be made with "1" as the Maximum Threshold, and a "-1" as the Minimum threshold. Although "-1" is an unachievable state, this triggers a notification when the Maximum Threshold is met, thus only notifying the NCAP Client when the Transducer moves to a True state.

This same transducer could also be used to alert a house owner when it changes states. If there is direct access to the TIM (if it is programmable), a StateChange variable can be created and have the TIM send an alert to the NCAP Server if this value changes. However, most users may not have access or the programming skill to be able to change the software running on the TIM. Another method would track the StateChange variable in the NCAP Server and initiate an alert if the state changes. This option is only available for programmable NCAP Servers. This leaves the NCAP Client, which is more easily accessible by a user. The NCAP Client would first send a SetupSensorAlertThreshold Request, changing the minimum threshold to "0" or "FALSE" and the maximum threshold to "1" or "TRUE". Doing this will send an alert to the NCAP Client when the state changes in either direction. The applications running on the NCAP Client can then analyze the alert and determine from the previous state of the Transducer if there was a change.

Transducer Management Services. The last set of services which an NCAP Server can provide to an NCAP Client are the Transducer Management Services. These services alone could be an entire thesis: This work acknowledges their existence but makes no further advancement. These services contain the ability to check the health of the entire network. Starting at the internet communication, there are services which allow the checking of packet loss rates, latencies, and link utilization between the NCAP Client and Server. At the TIM level, services including fault diagnostics, overall health reports, self-testing initiation, location information, and calibration settings are accessible. Each one of these services needs careful consideration based on the type of communication protocol, technology, and overall resources available to the TIM and the NCAP Server.

Problem Definition

The NCAP Server could be considered the keystone in the architecture of the IEEE 1451 Family of Standards. It needs to be able to communicate on the internet to the NCAP Client utilizing communication protocols such as UDP, XMPP, and more, potentially at the same time. The NCAP Server also needs to be able to communicate to multiple TIMs, each possibly using different communication methods varying from serial interfaces to wireless. For each of these types of TIM interfaces, the NCAP Server needs to have the proper drivers to operate on those interfaces with proper protocols to talk to specific TIMs. Since all TIM's are not the same, the NCAP Server needs to also keep track of which TIMs are connected at any given time. This can be done by learning enough information about them from the TEDS on the TIMs to be able to communicate with the correct protocol. The NCAP Server must also be able to handle multiple requests at the same time from potentially different NCAP Clients. Since machine and human health could be at risk in some implementations, the NCAP Server cannot simply just queue messages and wait for each one to process. Imagine an NCAP Server attempting to read 5 minutes’ worth of data from a sensor, and while this is happening, another sensor breaks a threshold indicating a fault in the system. Without proper care, that alert could take up to five minutes to propagate to those who need notifying.

One of the interesting goals of the IEEE P21451 Family of Standard is to allow a system designer to still create their own hardware and software but giving them guidelines so that they can easily integrate their work with an existing network. Instead of having to recreate the wheel in attempts to establish their own message structure and set of common services, they can instead use the standard as a framework to build their products around. This is the goal of many communication standards working groups. There are two common factors, which hinder the development and adaptation of any communication standard. First, the concepts, theory, and layout of the standard need to be explained in such a way that its intended uses are easy to understand. This includes defining new objects, abstractions, or key terms clearly so that someone looking to adopt the standard does not have to spend hours, days, weeks, or even months of a company’s time and money trying to understand how it works. Having to invest this much time and effort may turn away those who could benefit from a standard in lieu of making their own proprietary implementations or to adopt a different standard.

As mentioned previously, one designer demographic that could pave the way for an extensible and adaptable IoT are the Makers. Many websites and forums support the Maker community where ideas and questions are exchanged. One common theme is

Open-Source code, libraries, or designs. Due to the nature of the open-source materials that are used, designers have open access to see what exactly is happening in the code and can change certain pieces to fit to their needs. If the designer finds a bug in the code, a driver conflict, or maybe a better implementation of some function, they can send a request to the owner of the code, providing a means of user feedback.

The second feature which was found to be important in the adoption of a standard is the availability of reference designs and how many resources there are to help construct and test a compliant system. This is different from what has already been mentioned is that this feature deals specifically with the actual implementation of the standard. If the only reference material is the standard documentation itself, it can be extremely difficult to ensure that the implementation complies within the bounds of the standard. Within the realm of the IEEE 1451 Family of Standards, there are no reference designs that someone could purchase or design easily to test out if their interpretation of the standard is compliant. This means that if a designer has an idea to make a smart process and develop a TIM to facilitate connecting this process to the internet, they also must design from scratch an NCAP Server as well as the NCAP Client. This means way more cost, time, and skill than the designer may be able to put towards the project. On the other end of the spectrum, if someone has an idea for a smartphone application to interface with IEEE 1451 compliant networks, they will have to either find or create their own smart transducer network.

Three problems are proposed to be solved.

1. The NCAP Server object within the standard should be built on readily available, low cost embedded platforms.

2. Software and hardware developed during this research must be Open Source.
3. Reference designs must be developed for easily implementable devices.

Chapter 3

Methodologies and Project Management

This research was based on the work performed by a previous graduate student which began to implement a basic IEEE 1451 Smart Transducer Network. This implementation was focused on a single NCAP Server, single NCAP Client, and a single TIM. During the first semester of working on this project in the Spring of 2015, the project team consisted of 10 undergraduate students and 2 graduate students. The previous graduate student was using a shared DropBox folder to organize reference materials, drafts of the standards, as well as the code. The entire project team would meet twice a week during the Junior/Senior Clinic class times, where most of the development on the project was performed. While the previous methods of communication, resource and code management, and project management worked for a smaller group, a new management structure and project schedule was needed to facilitate larger groups of students. This, in combination with learning about the standard and reading through all the documentation related to the project, mainly comprised the first quarter of this research.

Familiarization With the Standard and the Current Implementation

In the literature, research related to the IEEE P21451-1 standard focused on a functionally limited NCAP Server, a basic NCAP Client, and a basic TIM. The first major research goal evaluated the existing implementation to determine compliance with the family of standards.

Initial NCAP Server Implementation. The initial NCAP Server had been developed to work on a Raspberry Pi Model 1 B+ development board, which can be seen in Figure 17. The Raspberry Pi (RPi) Model B+ development platform contains a Broadcom BCM2835 System-on-a-Chip (SoC), which houses a 700 MHz Low Power ARM1176J2FS Applications Processor, a dual core VideoCore IV® graphics co-processor, and 512MB of RAM. This model of the RPi also contains 4 USB 2.0 ports, a 10/100 Base T Ethernet Socket, a full HDMI female connector, microSD card slot, and 40 General Purpose Input/Output (GPIO) pins. Power is supplied via a micro USB cable, requiring a 5-volt power supply that can provide up to 2 Amps of current.

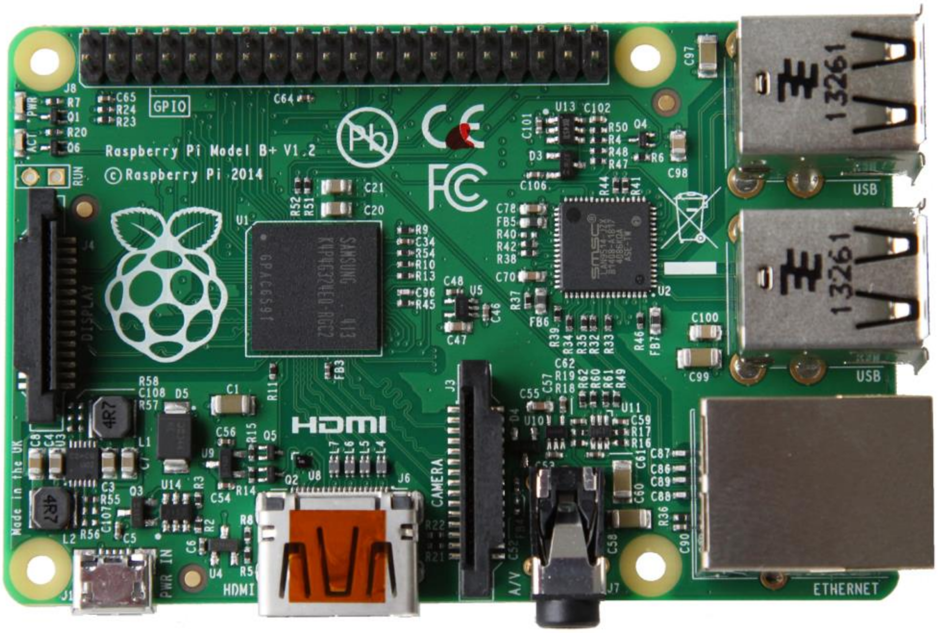


Figure 17. Raspberry Pi Model B+ ARM based development board

This development board was chosen by the previous graduate student due to the available amount of supporting documentation, open-source libraries, low cost, and the ease of use. The Raspberry Pi board uses a special version of Debian Linux called Raspbian, which allows users to treat this board as a stand-alone, Single-Board computer. Since the developer has access to a full operating system, one of the programming languages that is natively supported by the operating system is Python, which was chosen as the main programming language for the NCAP Server. This development board is also widely used around the world as an introduction to embedded systems and a tool for rapid prototyping.

Choosing a Programming Language. While Python at first glance may not seem like the best choice for an embedded system, due to the number of resources required to support programs and the fact that Python is not a compiled language, it allowed rapid development. This is simpler than corresponding tedious details associated with languages such as C. An example of this simplicity is summarized in the “Hello World” metric in Appendix A. Printing “Hello World” in common languages such as C, C++, and Java take between five to six lines of code to whereas Python only takes one.

Python also has large support for the Raspberry Pi, as the interpreter needed to run the code is built into the Raspbian operating system. Low-level platforms dealing with serial communications, hardware abstraction layers, and driver calls were previously dominated by the compiled languages C and C++. With the Python interpreter being directly linked to the operating system, it can now perform these same functions while providing high levels of functional abstraction. Because of this native support for Python, it has been one of the fastest growing embedded platforms, alongside platforms such as

Arduino, in the world of makers. As mentioned previously, where there are makers, there are tutorials, reference designs, and examples abound. This explosion in popularity also has caused those companies who curate products for makers to begin making Python libraries for their inventory. All of this comes together as a very compelling argument for using Python for an NCAP Server implementation; however, there must be an analysis of the negative impacts Python use could have on a system.

Drawbacks of Python. Along with the fast-paced, yet gentle, learning curve and high-level functional nature of the language, Python also has an incredible amount of resources in the forms of tutorials and forums, especially when using it with the Raspberry Pi. One of the requirements of the Raspberry Pi is an operating system, or at least a Python interpreter. This architecture requires substantial power, which limits an implementation from being exported to low-power microprocessors. As mentioned in the problem statement, this could be an issue if firmware is to be designed which can be ported to many different devices. Because of the interpreter, it also can be difficult to design a system that consumes small amounts of power during normal operation, as compared to compiled languages on processers which support sleeping modes. However, there are ways which some of the overhead power requirements within the Raspberry Pi can be mitigated, as discussed later in this section.

Having the requirement of the interpreter also can limit the speed of execution of a program. Since it is not compiled to machine language and executed at that level, it takes the interpreter time to perform the proper driver calls and function to achieve the same effect. This being a major argument for not choosing Python in many cases, developers have begun making interpreters such as PyPy which attempt to bring the speed and efficiency of language such as C by compiling the Python code into something closer resembling machine code. Using these “compiled” interpreters may require different syntax with certain function calls, and as such, the use of these interpreters in place of the native CPython interpreter was not looked at. Using alternate interpreters would be a future step in maximizing the performance of implementations on specific platforms.

One other major issue that is often cited is that Python code does not allow for the flexibility in coding style that many other languages offer. This is mainly due to the use of whitespace instead of brackets to separate sections of code. This means that if two people want to work on the same piece of code, they must choose how to indent their code so that during runtime, there are no issues with functional hierarchy. While there is no formal definition for spaces required to indent the lines within an enclosed function block (such as an if statement), the programmers must choose between using tabs or spaces when doing indentation. A related issue is that even though the programmers might utilize tabs for indentation, different machines treat tabs in different ways, making it difficult to sometimes move code from one person’s computer to another. While this can take a little getting used to and since most code editors insert spaces instead of the tab character, the result of forcing programmers to use indentation is human-readable code that is simple to follow and understand. This project consistently had 6-10 students working on it at any given time, making this issue quite a challenge to those who did not note which spacing convention others were using. Even with these concerns, it was found that Python was still the target language for an initial implementation.

Initial NCAP Server/TIM Hybrid Implementation. For simplicity, the implementation created by the previous graduate student combined the NCAP Server and the TIM abstraction layers into one device. Although this may seem like this goes against everything that has been talked about so far, there is nothing in the standard that prevents combining these two abstraction layers together. When combined, the NCAP Server must contain all the proper drivers as well as the functionalities which the TIM traditionally would contain. An advantage to this type of implementation is the ability for the NCAP Server services to have a much closer relationship with the TIM services, providing for quick execution and a reduction in overall system cost. Because the implementation used the Raspberry Pi, the 40 digital GPIO pins were available to interface with transducers and signal conditioning circuitry.

A block diagram of this implementation can be seen in Figure 3, with the NCAP Client being an XMPP or UDP chat client, and the NCAP Server and TIM being located on the Raspberry Pi. The three transducers which were chosen to be controlled were a thermistor, a fan, and a light. For signal conditioning, the thermistor was placed in a voltage divider configuration and was powered using the 5V rail of the Raspberry Pi. Since the GPIO lines of the Raspberry Pi are digital, a Texas Instruments ADS1015 12-bit ADC was used to obtain the analog voltage across the thermistor. Since the fan and the light both required 120V AC, solid-state relays were used to interface between these devices and two digital GPIO pins on the Raspberry Pi.

NCAP Server implemented functions and operation. The implementation was primarily focused on establishing communication between the NCAP Server and NCAP Client, as well as getting two major services to work:

ReadTransducerSampleDataFromAChannelOfATIM and WriteTransducerSampleDataFromASingleChannelOfATIM. It was decided that the Transducer Access Services could be constructed from a robust version of these two services. Since the initial implementation only required single reads and writes, it made sense to focus on developing these functions first as well. The previous work which had been done on the project relied upon using the User Datagram Protocol (UDP) as the way to communicate between NCAP Client and NCAP Server. With these requirements in mind, the procedural flow of the NCAP Server's program is as follows.

The first step the NCAP Server did upon being turned on and the program started with initializing the libraries required to communicate via UDP, including verifying a stable connection to the internet. During this time, the NCAP Server also registered itself as an NCAP on the network it was connected to by establishing an NCAP ID for itself. Once this was completed, the NCAP Server initialized the GPIO on the Raspberry Pi as well as the drivers required to communicate to the transducers. After all the proper initialization was completed, the NCAP Server would then be ready to receive messages from the NCAP Client. At this point, the NCAP Server was ready to receive and process messages, following the basic information flow diagram as seen in Figure 18.

In the example given in Figure 3, the NCAP Client sends a request for a single point of data from a transducer (in the case of the initial implementation, temperature from a thermistor). This message is received by the NCAP Server and then parsed based on the type of message received. At the front of the message is the Function ID, which tells the parsing function what the following data in the message pertains to. After parsing the required information, the proper communication drivers are called to communicate with the transducer, which in this case is handled over UART. Traditionally, the NCAP Server would not directly make a driver call for the Transducer, however, since this implementation is an NCAP Server TIM hybrid it must perform all the required tasks. Since only one point of data is required, once the data is ready, it is compiled and sent back to the original sender.

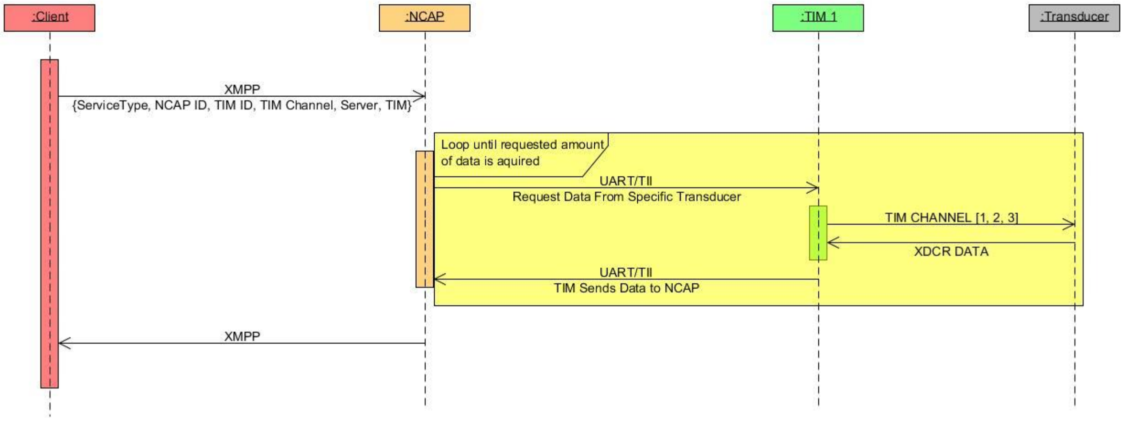

:Client
:NCAP
:TIM 1
:Transducer
XMPP
{ServiceType, NCAP ID, TIM ID, TIM Channel, Server, TIM}
Loop until requested amount of data is aquired
UART/TII
Request Data From Specific Transducer
TIM CHANNEL [1, 2, 3]
XDCR DATA
UART/TII
TIM Sends Data to NCAP
XMPP

Figure 18. Example communication between NCAP Client and Server requesting block data.

Chapter 4

Implementation, Results, and Discussion

“Low Cost” and Encapsulated Implementation

Room Monitoring TIM. Since the Room Monitoring TIM needed to be separate from the NCAP Server, the TIM implementation was migrated to the Texas Instruments MSP430F5529 Launchpad, as seen in Figure 19, once the functionality and initial performance were prototyped using a Raspberry Pi. This Launchpad can be purchased for less than $15 and contains a 16-bit MCU, 128KB of Flash Memory, 8KB of RAM, and can run up to 25MHz. This microprocessor also has an integrated 12-bit ADC, several timers, dedicated resources for communicating over serial communications and more. Because of the platform’s popularity, there is a lot of support to get different applications and transducers working with the device.

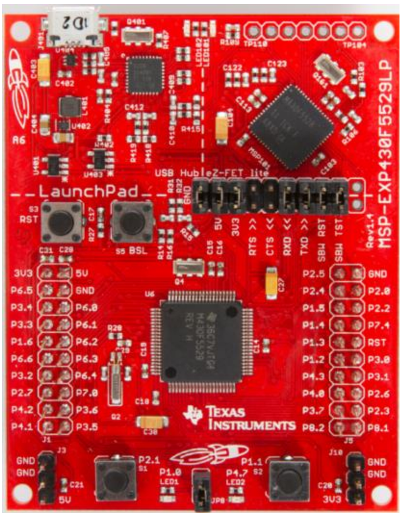


Figure 19. MSP430F5529 Launch Pad

The TIMs developed have two main components which need to work together: NCAP Server to TIM communication and transducer management. The development of these two subsystems was done initially with another Raspberry Pi so that the behavior of the TIM functioning on its own could be investigated. The transducer management subsystem which dealt with signal conditioning, acquisition, etc. was relatively straight forward to set up using a variety of transducers. The idea was to focus on building a TIM that was easily reproducible but also contain enough variety to show how to manage different types of sensors. For this, a 6 channel TIM was created which focused around an application in a Smart Building. A mixture of digital and analog sensors was selected along with a small LED array for actuation. A prototype of this TIM can be seen in Figure 20.

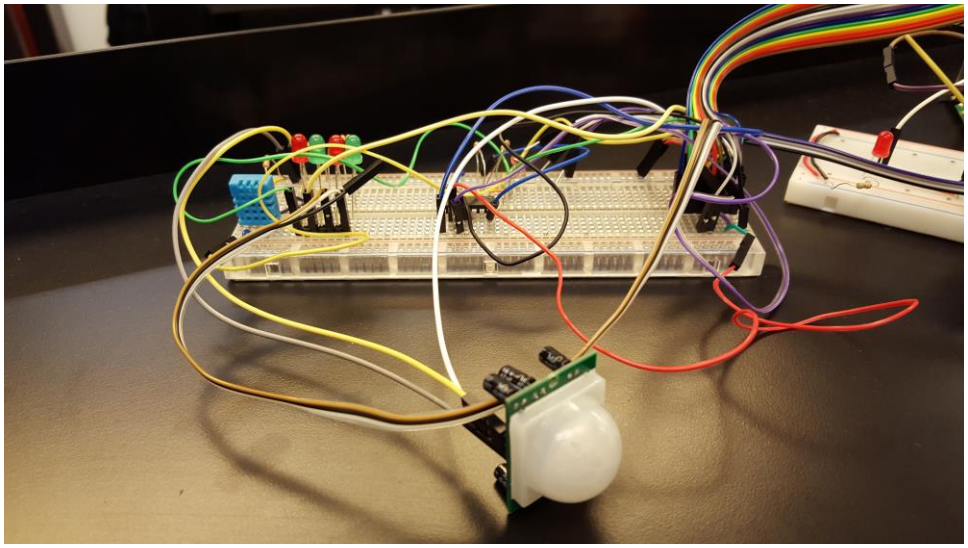


Figure 20: Prototype of a 6 Channel Smart Building TIM

The sensors utilized were a DHT11 Temperature and Humidity sensor, a Passive Infrared (PIR) sensor, a bulk thermistor, and a bulk photoresistor. Each of these sensors required differing levels of signal conditioning as well as protocols to retrieve the sensor information.

DHT11 Temperature and Humidity Sensor. The DHT11, as seen in Figure 21, is a low-cost digital sensor which works utilizing a 1-wire protocol to send the temperature and humidity to the requesting device in degrees and relative humidity. This sensor has an accuracy of roughly ±2°C for temperature and ±5% relative humidity, making it ideal for beginner application not requiring high accuracy. One caveat about this sensor is due to the protocol required to read data from the device, the device will return both temperature and humidity readings whenever data is requested. This meant that instead of treating this sensor as a single Sensor Channel, the temperature and humidity could be represented through two Sensor Channels. Both a single channel and dual channel implementation are supported by the IEEE 1451 Standards and provide certain benefits to the implementations of the NCAP Server and Client.

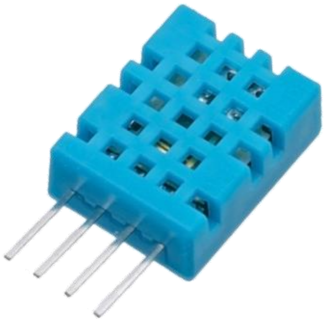


Figure 21. DHT 11 Temperature and Humidity Sensor

PIR Occupancy Sensor. The PIR sensor seen in Figure 22 is used to detect occupancy. The sensor contains an infrared sensor behind a pyroelectric lens, allowing for a viewing angle of around 120 degrees. By supplying 5V to the sensor, the onboard microcontroller and signal conditioning can determine whether there has been a change in occupancy in the room. If the sensor detects occupancy, it will raise the output pin high which can be read by the TIM; whereas no occupancy will bring the output low.

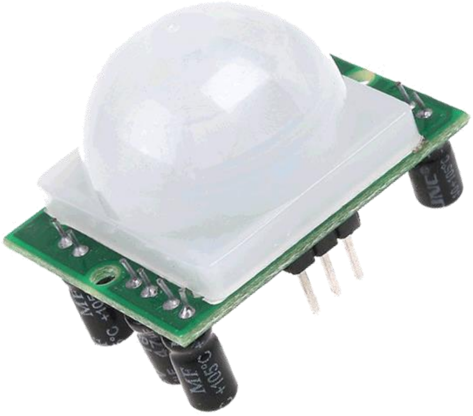


Figure 22. PIR Sensor

LED Array. An array of 4 LED’s with altering colors as seen in Figure 23 was used as the actuator for the TIM. There were two types of data which were experimented with to control which LEDs were on. The first method was sending an array of ones or zeros ({1,0,1,0}, for example) from the NCAP Client that would need to be parsed by the NCAP Server and sent to the TIM. While this allowed for intuitive control of the lights, it meant creating a new type of payload which would need to be handled by the WriteTransducerSampleDataFromAChannelOfATIM service within the NCAP Server.

To ensure that one “value” was returned by the function, the desired LED configuration such as {0,1,0,1} was represented as a hex byte, 0x05. The NCAP Client is responsible for converting the desired configuration into a corresponding byte and sending it to the NCAP Server. The TIM then converts the received byte into binary, which then assigns the proper state to each LED. Using this method also reduced the message complexity when requesting block writes.

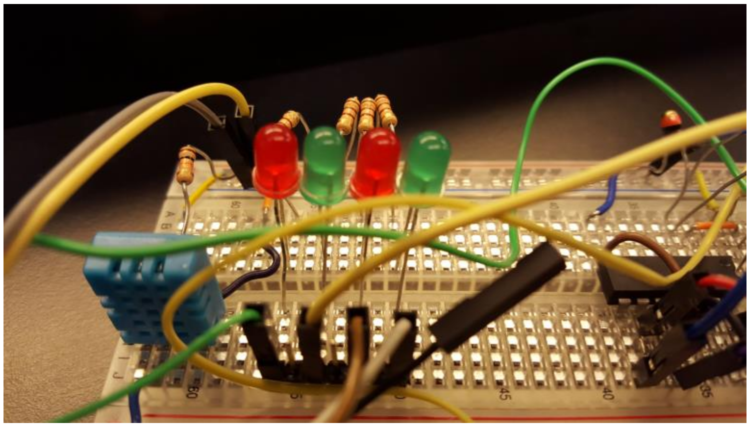


Figure 23. Green and Red LED Array

Photoresistor and Thermistor. The previously selected sensors had onboard signal conditioning and had a digital interface to retrieve the information. To show the extent of the mixed-signal capabilities of the standard, two analog sensors were implemented. To measure temperature, a Vishay 10kΩ nominal thermistor with 5% tolerance was utilized; light intensity was measured using a photoresistor. These sensors were placed into a voltage divider and its output was read by a 10-bit ADC. For the greatest amount of flexibility during implementation, the ADC readings were directly reported back to the NCAP Server which would then calculate the actual temperature or light level.

Ultrasonic TIM. A second TIM was implemented to test the services which interacted with multiple TIMs. The Elec Freaks HC-SR04 Ultrasonic Sensing Module, as seen in Figure 24, is another self-contained sensor requiring no signal conditioning other than power. The module works by the TIM holding the Trigger pin high for at least 10 microseconds, which tells the module to begin sending 40kHz pulses out towards the direction it is facing. The module then will listen for any pulses that would be reflected from an object. When it determines there is an echo, it will raise the Echo pin high. The distance can then be calculated using the time-of-flight for the pulse and how fast the pulse was traveling. The TIM is responsible for performing this calculation, and the results are what is returned to the NCAP Client.

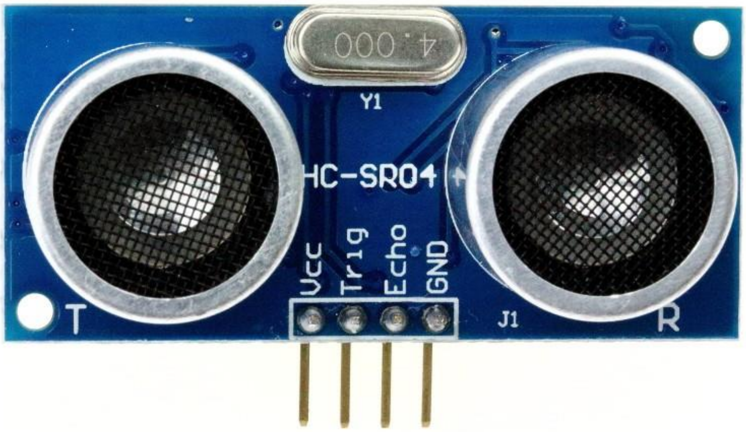


Figure 24. HC-SR04 Ultrasonic Sensor

This TIM implementation also allows for a simple test of the Event Notification services. Temperature and Light levels can be difficult to precisely control without specialized equipment, whereas distance from a sensor can be quickly measured and manipulated. This makes the Ultrasonic sensor a better candidate for testing these Event Notification services.

TIM Operation

Both TIMs at the software level operate in the same way, with the only difference being how they acquire sensor data and manage actuators in the "Transducer Management" layer. Above this layer is the "NCAP Server to TIM Communication" layer. This manages the communication between the two entities and is responsible for decompiling messages, making the correct driver call, and compiling message back to the NCAP Server. These fundamental functionalities, reading and writing to transducer channels, were isolated and implemented first since the research is oriented at the implementation and behavior of the NCAP Server.

NCAP Server – TIM Communication Structure. The interface between the NCAP Server and TIM was chosen to be UART. Messages sent from the NCAP Server to the TIM had the following format: "TIMFunctionID, TIMChannelID, \r" where \r is the carriage return byte in ASCII. Unlike the FunctionID mentioned before, these TIMFuctionIDs correspond to those found in Appendix B. The two TIMFunctionIDs which were implemented in both TIMs were 0 and 128.

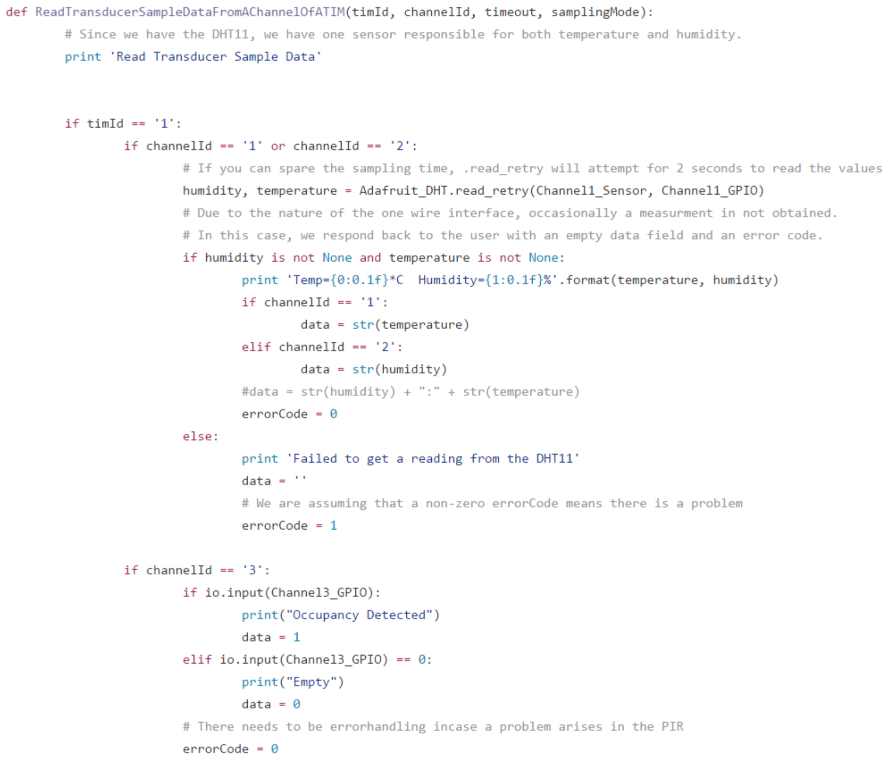
When an NCAP Client sends a request for information about a sensor, the NCAP Server parses the message from the NCAP Client and determines which TIM and Transducer Channels it needs to access. From here, based on if the operation is a Read or

Write, the message will be compiled in the previously mentioned form and sent to the TIM over UART. The TIM will then parse through the incoming message and identify which channel it needs to access. The appropriate driver calls are made to access the sensor information or write to the actuator. Following a successful execution will send back to the NCAP Client the data requested and/or an acknowledgment that the action completed. If the NCAP Server wants to read or write block data, it will need to send requests for individual data point until the required amount of data has been read/written.

Global vs Individual ChannelID. The TIMChannelID can take two different values, either a 0 or a specific transducer channel. If a non-zero TIMChannelID is received, whichever function is called will operate only on that corresponding channel. If a TIMChannelID of 0 is received, this changes the function to act on a Global scale, meaning that it will perform the function for all the available channels. For example, if an NCAP Server wanted to request readings from all the Transducer Channels on a 40-channel TIM monitoring a manufacturing process, the NCAP Server would need to send individual requests for every single transducer channel. This takes up a large amount of time on both the NCAP Server and TIM in compiling, sending, and parsing through loads of messages. Instead, the NCAP Server can send a request (for example, “128,000,/r”) to the TIM, and the TIM will then compile a message by concatenating the data from each channel starting at TIMChannelID 1, thus eliminating the superfluous addressing overhead.

Abstracting the TIM From the NCAP Server

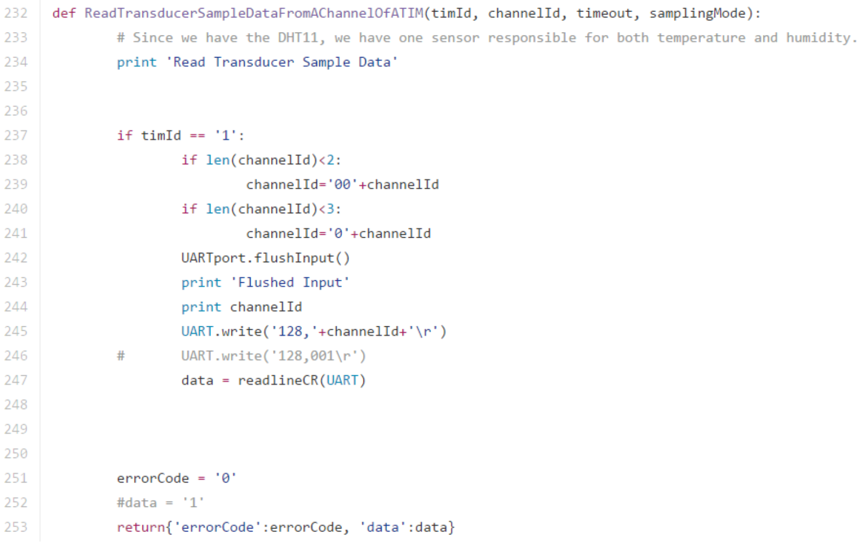
The implemented TIMs contained the signal conditioning circuitry and drivers, which previously were in the NCAP server. This led to restructuring the NCAP Server to operate solely on sending properly formatted messages to these TIMs. An example of how this is impacted the overall code can be seen in Figure 25, with long comments removed for readability.



```
def ReadTransducerSampleDataFromAChannelOfATIM(timId, channelId, timeout, samplingMode):
        # Since we have the DHT11, we have one sensor responsible for both temperature and humidity.
        print 'Read Transducer Sample Data'


        if timId == '1':
                if channelId == '1' or channelId == '2':
                        # If you can spare the sampling time, .read_retry will attempt for 2 seconds to read the values
                        humidity, temperature = Adafruit_DHT.read_retry(Channel1_Sensor, Channel1_GPIO)
                        # Due to the nature of the one wire interface, occasionally a measurment in not obtained.
                        # In this case, we respond back to the user with an empty data field and an error code.
                        if humidity is not None and temperature is not None:
                                print 'Temp={0:0.1f}*C  Humidity={1:0.1f}%'.format(temperature, humidity)
                                if channelId == '1':
                                        data = str(temperature)
                                elif channelId == '2':
                                        data = str(humidity)
                                #data = str(humidity) + ":" + str(temperature)
                                errorCode = 0
                        else:
                                print 'Failed to get a reading from the DHT11'
                                data = ''
                                # We are assuming that a non-zero errorCode means there is a problem
                                errorCode = 1

                if channelId == '3':
                        if io.input(Channel3_GPIO):
                                print("Occupancy Detected")
                                data = 1
                        elif io.input(Channel3_GPIO) == 0:
                                print("Empty")
                                data = 0
                        # There needs to be errorhandling incase a problem arises in the PIR
                        errorCode = 0
```

Figure 25. Code Snippet of an older version of a Single Transducer Read containing drivers.

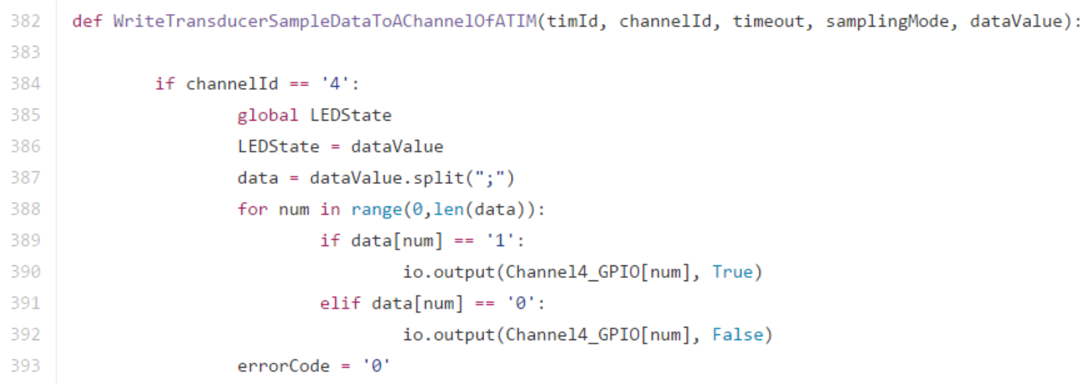
This code snippet (35 lines out of roughly 100 for the entire function) is just one service (reading a single point of data from a single transducer on a single TIM), and only covers 2 out of the already mentioned 5 sensors that are contained between the Room Monitoring and ultrasonic TIMs. Not only does including these driver calls inside of this function create non-reusable code, but having these drivers also slow down the execution of this function. The ReadSampleDataFromAChannelOfATIM function “Reading Work Horse” function since in the NCAP Server, the rest of the Transducer Read functions implemented call upon this function. For example, if block data is required to be read, the NCAP Server will in a loop call the Reading Work Horse function for however many points of data are required. It can be easy to see why making this function as minimal and efficient as possible it of upmost importance to the overall performance of the NCAP Server. By abstracting the driver calls into the TIMs themselves, that same Reading Work Horse function can be written in roughly 15 lines of code, as can be seen in Figure 26. In this version of the function, all that is required are simple string concatenations and a single driver call to send and listen to the TIMs over the UART connection. Not only is this code more legible to the average coder, but it is computationally simpler and less strenuous than the previous version.



```
def ReadTransducerSampleDataFromAChannelOfATIM(timId, channelId, timeout, samplingMode):
        # Since we have the DHT11, we have one sensor responsible for both temperature and humidity.
        print 'Read Transducer Sample Data'


        if timId == '1':
                if len(channelId)<2:
                        channelId='00'+channelId
                if len(channelId)<3:
                        channelId='0'+channelId
                UARTport.flushInput()
                print 'Flushed Input'
                print channelId
                UART.write('128,'+channelId+'\r')
        #       UART.write('128,001\r')
                data = readlineCR(UART)



        errorCode = '0'
        #data = '1'
        return{'errorCode':errorCode, 'data':data}
```

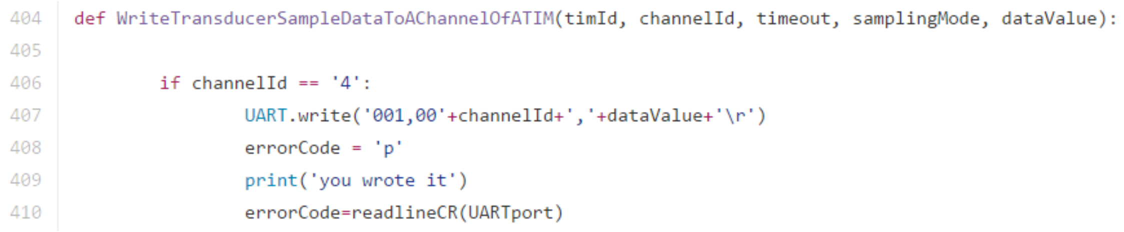
Figure 26. Code Snippet of the simplified Reading Work Horse function after removing TIM-specific

This same effect can be seen in the "Writing Work Horse" function, WriteSampleDataToAChannelOfATIM. The original function before the stand-alone TIMs can be seen in Figure 27. While the driving portion of this function is not as complex, it still requires the NCAP Server to run through a loop for each point of data and directly access the GPIO on the Raspberry Pi. Specifically, for the LED array, this also utilizes quite a large amount of GPIO pins, limiting how many other TIMs or transducers can be connected. Once the drivers were abstracted away, the Writing Work Horse function was simply implemented in a few lines as seen in Figure 28. Without the debugging code on lines 408 and 409, it can be seen that after checking to make sure the requested ChannelID can be written to, all the is required is compiling a UART message containing the data and the ChannelID that is passed into the function and sending it.



```python
def WriteTransducerSampleDataToAChannelOfATIM(timId, channelId, timeout, samplingMode, dataValue):

        if channelId == '4':
                global LEDState
                LEDState = dataValue
                data = dataValue.split(";")
                for num in range(0,len(data)):
                        if data[num] == '1':
                                io.output(Channel4_GPIO[num], True)
                        elif data[num] == '0':
                                io.output(Channel4_GPIO[num], False)
                errorCode = '0'
```

Figure 27. Code Snippet of the Writing Work Horse function before abstracting the TIM specific drivers.



```python
def WriteTransducerSampleDataToAChannelOfATIM(timId, channelId, timeout, samplingMode, dataValue):

        if channelId == '4':
                UART.write('001,00'+channelId+','+dataValue+'\r')
                errorCode = 'p'
                print('you wrote it')
                errorCode=readlineCR(UARTport)
```

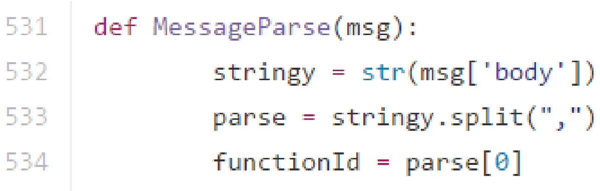
Figure 28. Code Snippet of the Writing Work Horse function after abstracting the TIM specific drivers.

NCAP Server Operation

There are a few main groups of services and functionality which make this implementation possible. First, the NCAP Server must connect with an XMPP server target so that it can begin to transmit and receive messages. Once connected to the XMPP server, the NCAP Server can begin managing NCAP Client subscriptions, processing requests, and connected TIMs. A few techniques such as the parsing function and threading will be discussed as they were important to solve earlier problems. The coding style was chosen to better support development of a framework for a library or Software Development Kit (SDK), which can be used to quickly and easily add Standard functionality to a project.

NCAP Server Initialization. The communication utilizing the XMPP server is managed on the Raspberry Pi by a library called SleekXMPP, which would need to be installed on the NCAP Server in any implementation utilizing this code. As previously mentioned, upon powering up and establishing an internet connection, the NCAP Server will attempt to connect to a remote XMPP server. This is done through an eXtensible Markup Language (XML) roster-based system. The Jabber IDs (JIDs) or usernames of all parties connected to an XMPP server will be shared with the NCAP Server. During this process, one of the messages the NCAP Server sends to the XMPP Server is a request to have its status changed from Offline to Online, indicating to those entities who are subscribed to the NCAP Server that it has turned on. Once this has taken place, the NCAP Server then loads a set of XMPP extension protocols (XEPs) which are required by the P21451-1-4 supporting document for P21451-1. After these XEPs are loaded from the XMPP Server, the Raspberry Pi begins to run in a loop listening for any incoming messages sent by the XMPP Server.

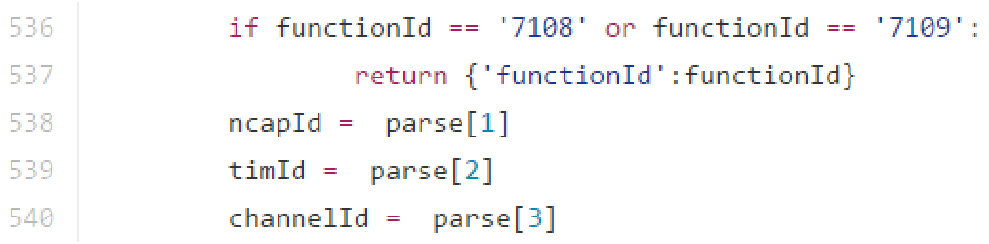
Receiving and Parsing Messages. Once the NCAP Server is ready to listen, a callback routine is utilized so that whenever a message is received, the information can be extracted and passed on to other functions. The first function called during the callback is MessageParse(msg), where msg is the actual XML stanza sent from the NCAP Client. The first part of the MessageParse function, as seen in Figure 29, is isolating the body of the message which contains all the important addressing information as well the all-important FunctionID.



```python
def MessageParse(msg):
        stringy = str(msg['body'])
        parse = stringy.split(",")
        functionId = parse[0]
```

Figure 29. Code Snippet of the initial parsing within the MessageParse() function.

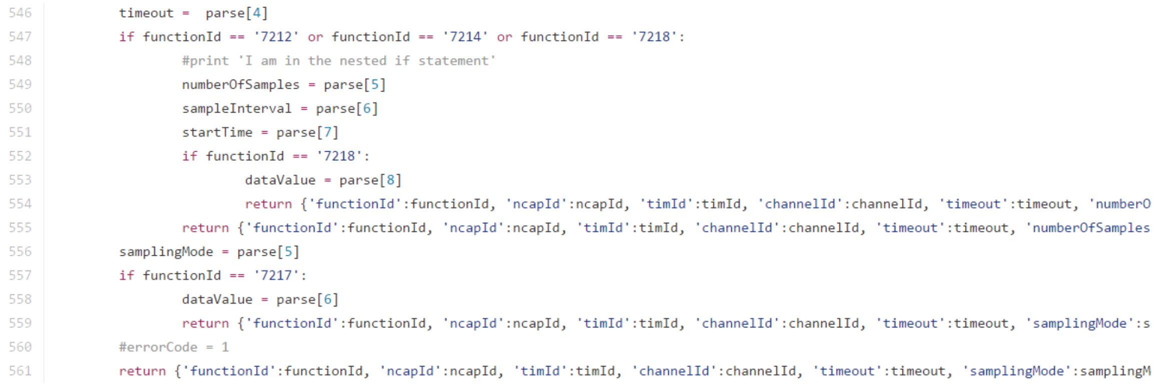
The body of the message is comma delimited, meaning that all the fields for each type of message are separated by commas. This string is parsed into an array with each cell informing the NCAP Server what to do next. The only common piece of information that is in every service available to the NCAP Client is that the first part of the body is always the FunctionID. As seen in Figure 30, the ClientJoin and ClientUnJoin services (FunctionIDs 7108 and 7109 respectively) only require the FunctionID and nothing else. If the service requested is one of these, the MessageParse function will stop immediately and return only the FunctionID back to the Callback routine. Otherwise, the other implemented services require NCAPID, TIMID, and ChannelId fields. The arguments for each of these services appear in the same order in the message body.



```
        if functionId == '7108' or functionId == '7109':
                return {'functionId':functionId}
        ncapId =  parse[1]
        timId =  parse[2]
        channelId =  parse[3]
```

Figure 30. Code Snippet of the MessageParse function after the initial parse.

Each of these services requires different information to operate. Figure 31 shows the parsing function. Depending on which service is requested, the information within a message varies based on membership and order. On return, the Callback resumes execution. Python does not have support for case-select structures; therefore, cascaded if statements are used to determine which function to call. This can be seen in Figure 32. Based on the FunctionID, the parsed information is passed to the corresponding service. In the original implementation, which also utilized UDP, two requests could not be processed in parallel. This was because the code was written to run sequentially, which is traditionally used in lower power microprocessors, leading to large delays in the completion of multiple tasks. One of the biggest advantages of the Raspberry Pi is that it runs a full version of Linux, which provides access to tools such as threading.

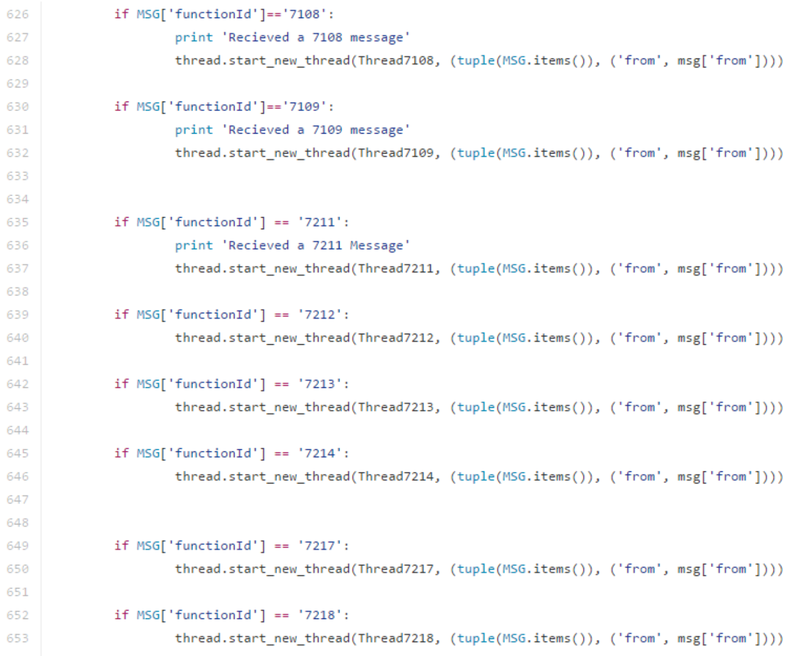


```
timeout =  parse[4]
if functionId == '7212' or functionId == '7214' or functionId == '7218':
        #print 'I am in the nested if statement'
        numberOfSamples = parse[5]
        sampleInterval = parse[6]
        startTime = parse[7]
        if functionId == '7218':
                dataValue = parse[8]
                return {'functionId':functionId, 'ncapId':ncapId, 'timId':timId, 'channelId':channelId, 'timeout':timeout, 'numberO
        return {'functionId':functionId, 'ncapId':ncapId, 'timId':timId, 'channelId':channelId, 'timeout':timeout, 'numberOfSamples
samplingMode = parse[5]
if functionId == '7217':
        dataValue = parse[6]
        return {'functionId':functionId, 'ncapId':ncapId, 'timId':timId, 'channelId':channelId, 'timeout':timeout, 'samplingMode':s
#errorCode = 1
return {'functionId':functionId, 'ncapId':ncapId, 'timId':timId, 'channelId':channelId, 'timeout':timeout, 'samplingMode':samplingM
```

Figure 31. Code Snippet of the MessageParse function which isolates specific information based on the FunctionID.

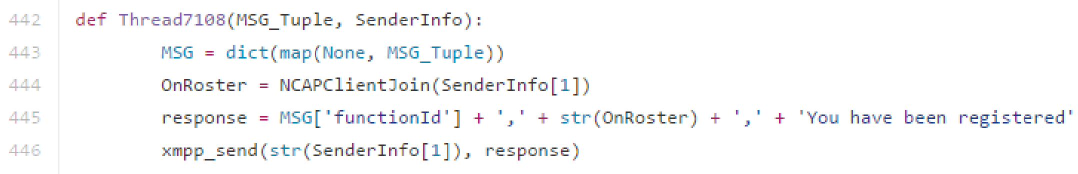
Threading. A thread is an encapsulated execution unit which the operating system can supply to that unit its own stack, set of registers, and a program counter. This allows parallel execution of code by breaking up the program into small, lightweight processes, which the operating system can quickly switch between. Using a native threading package for Python and Linux, the “Start_new_thread” command is called to begin a self-contained execution environment for each request. The design challenge at this point becomes determination of thread placement within the system in such a way as to minimize indeterminate states when dealing with multiple messages.

If every incoming message is assigned its own thread, the processing delay between each incoming message is reduced to the minimum amount of time it takes to parse it. This can be seen in Figure 32. After the parsing function is called, the body of the message and the addressing information are passed into a new thread. The Start_New_Thread call contains the previously specified services and the functionality needed to send a response. An example is shown in Figure 33.



```
        if MSG['functionId']=='7108':
                print 'Recieved a 7108 message'
                thread.start_new_thread(Thread7108, (tuple(MSG.items()), ('from', msg['from'])))

        if MSG['functionId']=='7109':
                print 'Recieved a 7109 message'
                thread.start_new_thread(Thread7109, (tuple(MSG.items()), ('from', msg['from'])))


        if MSG['functionId'] == '7211':
                print 'Recieved a 7211 Message'
                thread.start_new_thread(Thread7211, (tuple(MSG.items()), ('from', msg['from'])))

        if MSG['functionId'] == '7212':
                thread.start_new_thread(Thread7212, (tuple(MSG.items()), ('from', msg['from'])))

        if MSG['functionId'] == '7213':
                thread.start_new_thread(Thread7213, (tuple(MSG.items()), ('from', msg['from'])))

        if MSG['functionId'] == '7214':
                thread.start_new_thread(Thread7214, (tuple(MSG.items()), ('from', msg['from'])))


        if MSG['functionId'] == '7217':
                thread.start_new_thread(Thread7217, (tuple(MSG.items()), ('from', msg['from'])))

        if MSG['functionId'] == '7218':
                thread.start_new_thread(Thread7218, (tuple(MSG.items()), ('from', msg['from'])))
```

Figure 32. Code Snippet of the Callback Routine where threads are started based on the FunctionID.

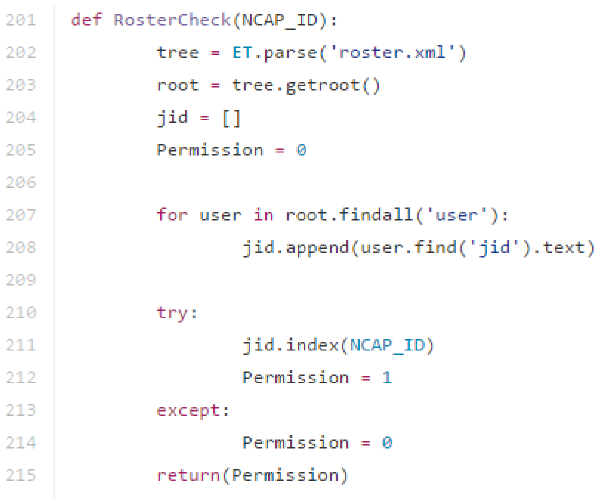


```
def Thread7108(MSG_Tuple, SenderInfo):
        MSG = dict(map(None, MSG_Tuple))
        OnRoster = NCAPClientJoin(SenderInfo[1])
        response = MSG['functionId'] + ',' + str(OnRoster) + ',' + 'You have been registered'
        xmpp_send(str(SenderInfo[1]), response)
```

Figure 33. Code Snippet of the function which runs inside a thread.

Each of the implemented services will need one of these threading functions. The first thing this function does is translate the message information from a tuple to a dictionary. Dictionaries are data structures much like an array; however, they have the added benefit of being searchable for labeled data. The requested service is called with input arguments filled with information in the converted dictionary. Depending on the service, information such as error codes and/or data is then returned to a thread-local variable. A response message is compiled and sent with this data and the NCAP Client identification information, and the thread is closed. Other threads can simultaneously execute.

Implemented Identification Services. While the XMPP Server handles most of the Discovery and Identification services laid out in the P21451-1 document, two main functions needed for a proof of concept was the NCAP Client Join and Unjoin services. These services manage an XML roster which contains all the JIDs of NCAP Clients which are subscribed to that NCAP Server. This lays a lot of the groundwork down for further work in utilizing the group services as well as implementing secure services. The way these services work is simple. For the Joining service, it checks the roster to make sure the NCAP Client is not already registered and then appends them to the list. Unjoin services will first check the roster against the transmitted JID and remove matching JIDs from the roster. The last function created from these services is a RosterCheck function, which can be utilized by other services. The RosterCheck function, as seen in Figure 34, simply takes in an NCAP Client ID and attempts to find a match in the roster. If it is successful, it returns a 1. Otherwise, the JID index function will fail. The try/except structure will catch and handle this condition by returning a 0.



```
def RosterCheck(NCAP_ID):
        tree = ET.parse('roster.xml')
        root = tree.getroot()
        jid = []
        Permission = 0

        for user in root.findall('user'):
                jid.append(user.find('jid').text)

        try:
                jid.index(NCAP_ID)
                Permission = 1
        except:
                Permission = 0
        return(Permission)
```

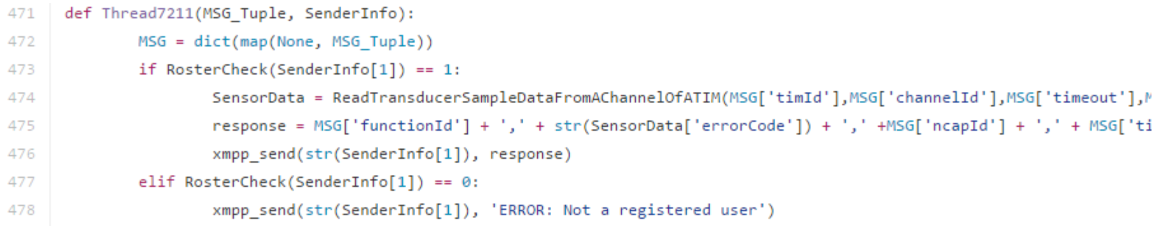
Figure 34. Code Snippet of RosterCheck function.

Implemented Transducer Access Services. At the core of the NCAP Server is the ability to acquire data from Transducers which are connected to TIMs. This meant that the following two services were the focus: ReadTransducerSampleDataFromAChannelOfATIM and WriteTransducerDataToAChannelOfATIM, which previously were referred to as the “Work Horse” functions of the Transducer Access Services. Other services within this group, such as ReadTransducerBlockDataFromAChannelOfATIM, calls back to the ReadTransducerSampleDataFromAChannelOfATIM service multiple times. Because of this, these two services need to be robust as well as easy to understand. These services will be discussed in two main groups, Reading and Writing.

Transducer Read Services. The “Workhorse” read function discussed previously and shown in Figure 26, contains the necessary concatenation functions to compile messages for UART-enabled TIMs. The “Workhorse” function needs a TIMID, ChannelId, timeout, and samplingMode. In this current implementation, different samplingModes and robust timeout functions were not implemented. The focus was on the communication model. The TIMID and ChannelId are extracted from the parsed information obtained from the original message.

The implemented network contained two different TIMs, so the TIMID determines which UART bus to utilize. Although communication is the same for each TIM, a generic approach expandable to multiple TIMs with multiple types of communication techniques was developed. A TIM roster is implemented, which has information collected from the TEDS onboard the TIM that defines communication protocols. The service would first check the timId against this roster to determine the method of communication (UART, I2C, Bluetooth, ZigBee, etc.). This approach allows for more autonomy within the network and creates more versatile NCAP Servers. The data retrieved through normal operations along with any error codes (for this implementation, default is 0) is packaged into a dictionary and returned to the main Thread function. The Thread function can package this information into a proper XMPP message and send it back to the NCAP Client.

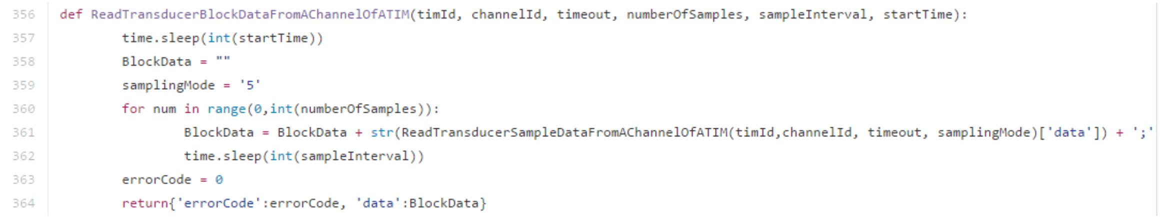
An additional feature added to the Reading Work Horse function was verification that the requesting NCAP Client was subscribed to the NCAP Server. The RosterCheck function validates this subscription before beginning to acquire any information. This can be seen in Figure 35. If the SenderInfo matches what is on the roster, the thread begins acquisition. Otherwise, it sends a message to the NCAP Client saying that they are not registered to the NCAP Server. Further work on the project should be the improvement of NCAP Server security. This can be done within the roster and roster check functions by adding elements from public-private key systems to secure messages over insecure channels.



```
def Thread7211(MSG_Tuple, SenderInfo):
        MSG = dict(map(None, MSG_Tuple))
        if RosterCheck(SenderInfo[1]) == 1:
                SensorData = ReadTransducerSampleDataFromAChannelOfATIM(MSG['timId'],MSG['channelId'],MSG['timeout'],M
                response = MSG['functionId'] + ',' + str(SensorData['errorCode']) + ',' +MSG['ncapId'] + ',' + MSG['ti
                xmpp_send(str(SenderInfo[1]), response)
        elif RosterCheck(SenderInfo[1]) == 0:
                xmpp_send(str(SenderInfo[1]), 'ERROR: Not a registered user')
```

Figure 35. Code Snippet of the Thread based function for the ReadSampleDataFromAChannelOfATIM service.

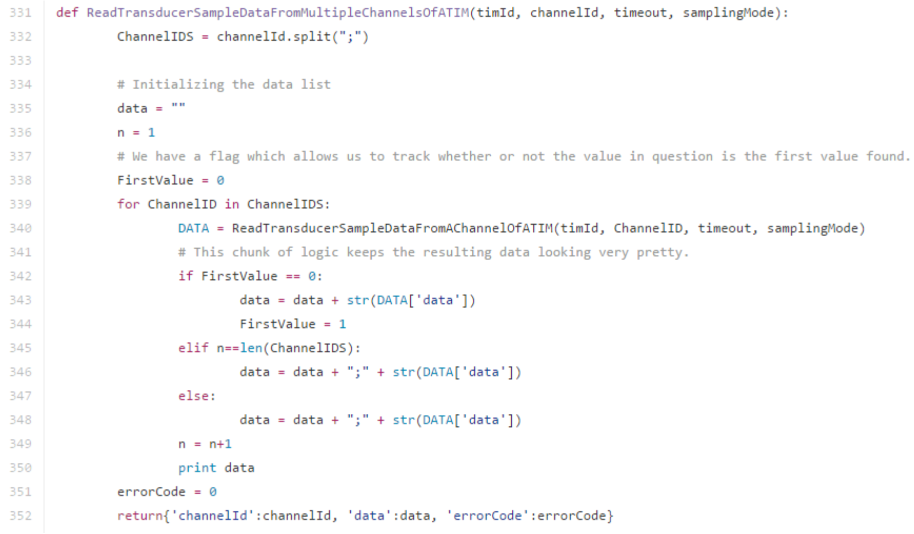
With the Work Horse function established, other Read services, such as ReadTransducerBlockDataFromAChannelOfATIM, can be constructed. An example is shown in Figure 36. Since the BlockData services require a start time, a local OS-based sleep function is used to pause the thread for a specific amount of time. The Reading Work Horse function is then called repeatedly until a specified number of samples are read. To enforce the SamplingInterval required for BlockData, the thread is paused at the end of every iteration of the for loop. For more precise collection of data, this service should be executed at the TIM level. Once there is the ability to obtain block data, there needs to be the ability to request data from multiple channels of a TIM at the same time. The major difference between the Single Channel Single Read service and the Multiple Channel Single Read is the use of multiple ChannelIDs. The first step of this service is to parse the ChannelIDs from a string into an array, as seen in Figure 37. The ChannelId string is delimited using semicolons to allow the parsing functions to properly operate.



```python
def ReadTransducerBlockDataFromAChannelOfATIM(timId, channelId, timeout, numberOfSamples, sampleInterval, startTime):
        time.sleep(int(startTime))
        BlockData = ""
        samplingMode = '5'
        for num in range(0,int(numberOfSamples)):
                BlockData = BlockData + str(ReadTransducerSampleDataFromAChannelOfATIM(timId,channelId, timeout, samplingMode)['data']) + ';'
                time.sleep(int(sampleInterval))
        errorCode = 0
        return{'errorCode':errorCode, 'data':BlockData}
```

Figure 36. Code Snippet of the ReadTransducerBlockDataFromAChannelOfATIM.

This corresponds to the chosen hierarchy of commas (,), semicolons (;), and then colons (:). This is done so that functions which need arrays can have that information passed through different parsing stages. From here, the Reading Work Horse function is called multiple times in a for loop much like with the block data; however, each time the workhorse function is called the ChannelID is incremented. The resulting data is compiled into a semicolon-delimited string.



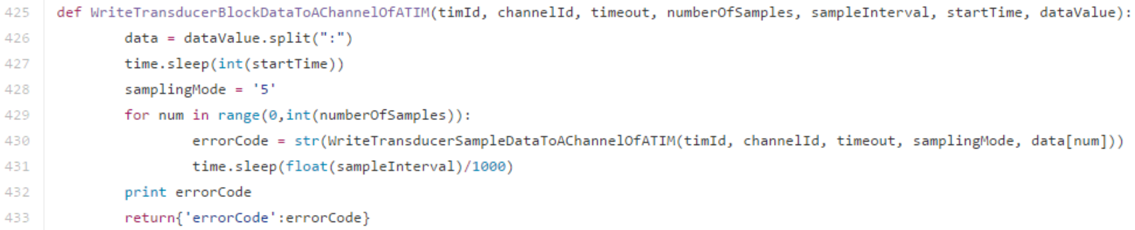
```
def ReadTransducerSampleDataFromMultipleChannelsOfATIM(timId, channelId, timeout, samplingMode):
        ChannelIDS = channelId.split(";")

        # Initializing the data list
        data = ""
        n = 1
        # We have a flag which allows us to track whether or not the value in question is the first value found.
        FirstValue = 0
        for ChannelID in ChannelIDS:
                DATA = ReadTransducerSampleDataFromAChannelOfATIM(timId, ChannelID, timeout, samplingMode)
                # This chunk of logic keeps the resulting data looking very pretty.
                if FirstValue == 0:
                        data = data + str(DATA['data'])
                        FirstValue = 1
                elif n==len(ChannelIDS):
                        data = data + ";" + str(DATA['data'])
                else:
                        data = data + ";" + str(DATA['data'])
                n = n+1
                print data
        errorCode = 0
        return{'channelId':channelId, 'data':data, 'errorCode':errorCode}
```

Figure 37. Code Snippet of the

For more complex functions such as "Read Transducer Block Data From Multiple Channels Of A TIM" or even "Read Transducer Block Data From Multiple Channels Of Multiple TIMs", the previously made functions can be repeaditly called on as shown before. As can be seen, getting the "Read Sample Data From A Channel Of A TIM" function as robust and efficient as possible is vital to the overall performance of the NCAP Server. By structuring the code in this manner, any hardware specific code for devices such as Bluetooth radios, serial lines, etc., can be abstracted away, making the process of adapting this code to other projects much less complicated.

Transducer Write Services. The writing workhorse function is the "Write Sample Data To A Channel Of A TIM" service, which can be seen in Figure 28. Just like with the Reading services, this workhorse can be called repeatedly within for loops to obtain the full list of writing services defined in the standard. The limitation of one TIM with an actuator meant the workhorse function and the "Write Block Data To A Channel Of A

TIM” were implemented. This is illustrated in Figure 38. For this data to be maintained as an array through parsing, the comma delimiter was utilized to separate the values which to be written. In the Write Block Data service, instead of stepping through a list of channelIds or the timIds, the data supplied are iterated through and those values are written. The only data returned from this function is an errorCode, which under normal operation should be 0.



```
def WriteTransducerBlockDataToAChannelOfATIM(timId, channelId, timeout, numberOfSamples, sampleInterval, startTime, dataValue):
        data = dataValue.split(":")
        time.sleep(int(startTime))
        samplingMode = '5'
        for num in range(0,int(numberOfSamples)):
                errorCode = str(WriteTransducerSampleDataToAChannelOfATIM(timId, channelId, timeout, samplingMode, data[num]))
                time.sleep(float(sampleInterval)/1000)
        print errorCode
        return{'errorCode':errorCode}
```

Figure 38. Code Snippet of the WriteTransducerBlockDataToAChannelOfATIM service.

IEEE SAS 2017: Plugfest

During the Spring semester of 2017, Rowan University hosted the IEEE Sensors Application Symposium where one of the events was an IEEE 1451 “Plugfest”. The idea behind a plugfest originally was so that people interested in the standard and that have developed NCAP Clients, NCAP Servers, or TIMs could come and interface with an established network. A tutorial was designed to guide participants in the use of the developed code base and the Raspberry Pi to develop standard compliant devices. For this tutorial, a few different resources were required.

To start, standard-compliant hardware for the participants was needed. For the

NCAP Server, the existing code base generated over the past semesters and the newer versions of the Raspberry Pi Zero that include WiFi and Bluetooth capability were used. The bigger issue at the time was the development of an example TIM that could be used to teach the standard. Due to the time constraints of the PlugFest workshop and no requirement on participants knowing how to program microcontrollers, the Raspberry Pi were again utilized for this. The Raspberry Pi having all digital I/O does not come with native sensors which could be utilized, so a “Pi Hat” or daughter card for use with the workshop had to be generated.

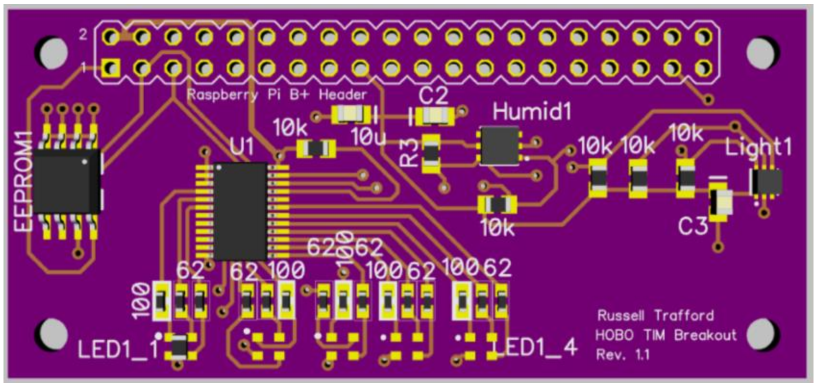
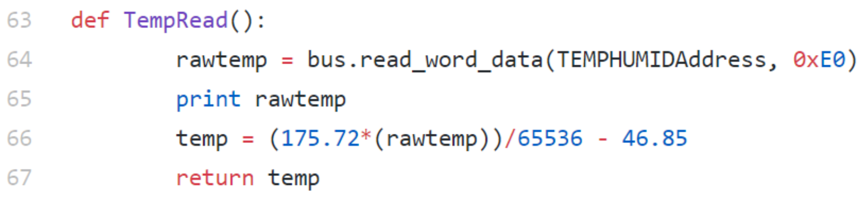


Figure 39: Building TIM used for SAS2017 Plugfest

The Building TIM seen in Figure 39 features an RM24C32DS EEPROM for TEDS and data storage, an SI 7006 Humidity and temperature sensor, an LTR-303 ambient light sensor, and five RGB LEDs controlled by a PCA9532 LED Driver. All these devices are connected back to the Raspberry Pi B+ style header, drawing power from the 3.3V and 5V lines provided by the Raspberry Pi, and communicating over a shared I2C bus. The bill of materials and the associated design files are in Appendix C. By utilizing I2C, generating the associated TIM code to read and write to these devices became very simple. The NCAP Server and the TIM were connected using UART for simplified testing, rather than implementing a wireless interface.

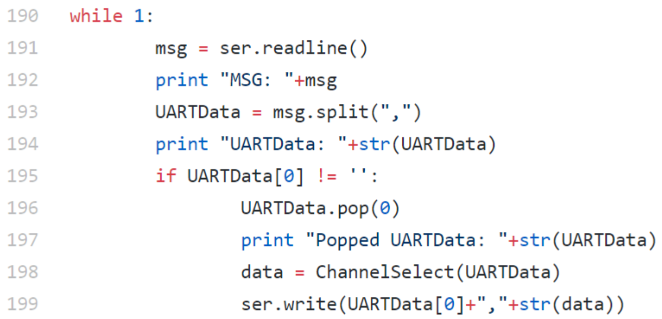
The software running on the TIM focused on the core functions as mentioned earlier in the main implementation: the ability to read sensors and write to them. When developing the code, encapsulating the sensor specific content for the I2C commands within their own functions became a primary focus. An example sensor function of this can be seen in Figure 40. Within this function, the I2C address and device specific command to read the temperature. This specific module will return a word which needs to them be converted into a temperature in Celsius.



```
def TempRead():
        rawtemp = bus.read_word_data(TEMPHUMIDAddress, 0xE0)
        print rawtemp
        temp = (175.72*(rawtemp))/65536 - 46.85
        return temp
```

Figure 40. Code Snippet of the Temperature Read function of the Building TIM.

The flow of the TIM program is as follows. After initializing all the sensors and the necessary drivers, the TIM awaits messages over the UART channel, as seen in Figure 41. Once a message is received, it is parsed using a comma as the delimiter. The Channel ID is then extracted from the message and passed into the ChannelSelect function.

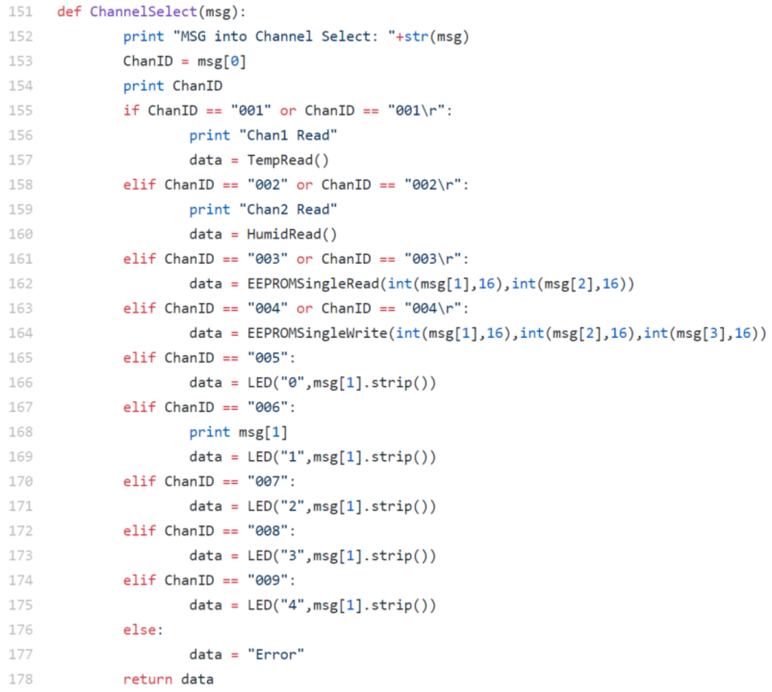


```
while 1:
        msg = ser.readline()
        print "MSG: "+msg
        UARTData = msg.split(",")
        print "UARTData: "+str(UARTData)
        if UARTData[0] != '':
                UARTData.pop(0)
                print "Popped UARTData: "+str(UARTData)
                data = ChannelSelect(UARTData)
                ser.write(UARTData[0]+","+str(data))
```

Figure 41. Main while loop for the Building TIM

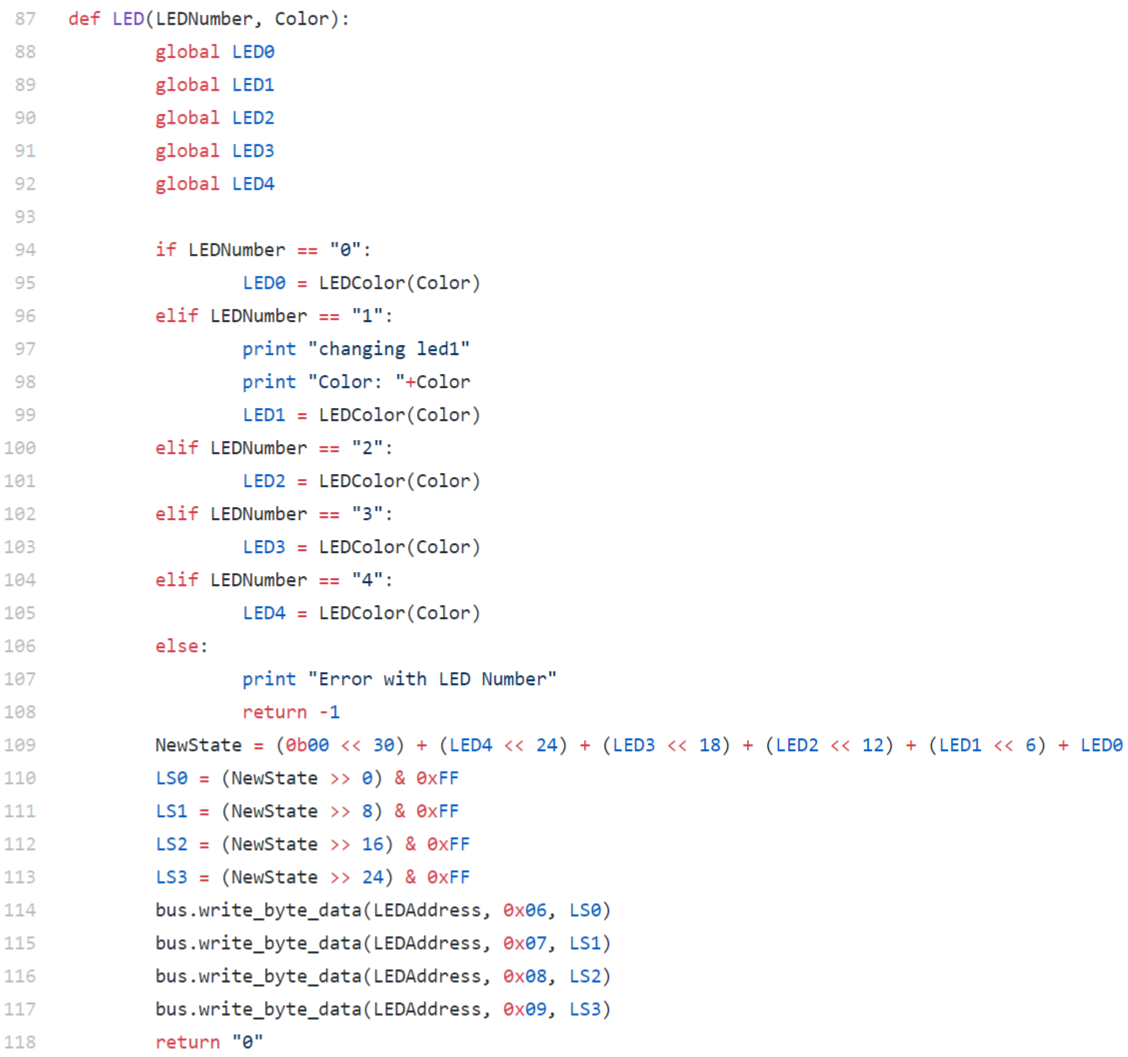
The ChannelSelect function seen in Figure 42 takes in the ChannelID from the NCAP Server message then calls upon the specific function to read or write the data. This function must have the ability to take in more than one string however because transducers which need to be written to require a value at which to set it at. The difference in how this function handles these two types of requests can be seen when comparing a call for TempRead() and a call for LED().

As previously mentioned, the TempRead function shown in Figure 40 will request a single temperature reading from the Humidity sensor. An example transducer access function such as setting the LED array state is shown in Figure 43. The LEDNumber and Color determine the bytes, which need to be sent to the LED driver. Since the LED driver works on 8-bit control registers for I/O, changing one LED requires an update to an entire register.



```
def ChannelSelect(msg):
        print "MSG into Channel Select: "+str(msg)
        ChanID = msg[0]
        print ChanID
        if ChanID == "001" or ChanID == "001\r":
                print "Chan1 Read"
                data = TempRead()
        elif ChanID == "002" or ChanID == "002\r":
                print "Chan2 Read"
                data = HumidRead()
        elif ChanID == "003" or ChanID == "003\r":
                data = EEPROMSingleRead(int(msg[1],16),int(msg[2],16))
        elif ChanID == "004" or ChanID == "004\r":
                data = EEPROMSingleWrite(int(msg[1],16),int(msg[2],16),int(msg[3],16))
        elif ChanID == "005":
                data = LED("0",msg[1].strip())
        elif ChanID == "006":
                print msg[1]
                data = LED("1",msg[1].strip())
        elif ChanID == "007":
                data = LED("2",msg[1].strip())
        elif ChanID == "008":
                data = LED("3",msg[1].strip())
        elif ChanID == "009":
                data = LED("4",msg[1].strip())
        else:
                data = "Error"
        return data
```

Figure 42. Channel Select function which calls maps the ChannelID's with specific Transducers.



```
def LED(LEDNumber, Color):
        global LED0
        global LED1
        global LED2
        global LED3
        global LED4

        if LEDNumber == "0":
                LED0 = LEDColor(Color)
        elif LEDNumber == "1":
                print "changing led1"
                print "Color: "+Color
                LED1 = LEDColor(Color)
        elif LEDNumber == "2":
                LED2 = LEDColor(Color)
        elif LEDNumber == "3":
                LED3 = LEDColor(Color)
        elif LEDNumber == "4":
                LED4 = LEDColor(Color)
        else:
                print "Error with LED Number"
                return -1
        NewState = (0b00 << 30) + (LED4 << 24) + (LED3 << 18) + (LED2 << 12) + (LED1 << 6) + LED0
        LS0 = (NewState >> 0) & 0xFF
        LS1 = (NewState >> 8) & 0xFF
        LS2 = (NewState >> 16) & 0xFF
        LS3 = (NewState >> 24) & 0xFF
        bus.write_byte_data(LEDAddress, 0x06, LS0)
        bus.write_byte_data(LEDAddress, 0x07, LS1)
        bus.write_byte_data(LEDAddress, 0x08, LS2)
        bus.write_byte_data(LEDAddress, 0x09, LS3)
        return "0"
```

Figure 43. LED Color Changing function of the Building TIM

Once the function execution has completed, control is passed back to the main while loop where it transmits a response to the NCAP Server. The NCAP Server, in turn, takes the data and translates it into a form useable by the XMPP server and the Client.

The overall flow of the PlugFest workshop started by familiarizing participants with the Raspberry Pi platform and the fundamentals of the IEEE P21451 architecture. The participants were then asked to work with two Raspberry Pi boards as if they were doing this at home and asked to command the Pi to pull the latest version of the TIM and NCAP code onto two different boards. From there, the daughter boards and appropriate connectors were distributed, and the participants were asked to experiment with the TIM code by utilizing USB-to-UART cables and a local serial terminal on their laptops. Once the felt that they understood how the TIM worked and the required message structure, they then connected their second Raspberry Pi to the UART channel of the TIM and began exploring the NCAP Server code. The workshop concluded by utilizing an XMPP chat client to send messages to the NCAP Server and seeing the results unfold on the daughter board.

Chapter 5

Conclusions

This chapter serves as a review of all the topics covered within this document and lay a groundwork for those building their systems modeled after the approach taken here. The motivation for and accomplishments of this research are briefly recapped. The implementation details are described, followed by recommendations for future work.

Summary of Research Accomplishments

The overall objective for this thesis was to take a close look at the IEEE P21451 Family of Standards, specifically the P21451-1 standard, and determine its suitability for IoT applications. The previous work of the Rowan University S.M.A.R.T. Lab defined the core services required of Smart Transducer Network for most use-cases. Implementation of each layer of abstraction (NCAP Client, NCAP Server, and TIM) was developed and matured. This approach met the objectives defined in Chapter 2 which are repeated below.

Low-cost and Easily Implementable NCAP Server. The Raspberry Pi platform was utilized as the prototyping platform, and because of this choice, Python was chosen to be the main programming language. Python allowed the code produced to not only be easily readable by humans but also to be easily implemented by other people. The Raspberry Pi also utilizes a full Linux Operating System, which allows us to leverage Threading to alleviate previous implementation issues such as system lag when processing multiple messages. All the code generated during the project was placed on GitHub®, paving the path to an open-source library of code for anyone to utilize. This repository, which is compatible with low-cost ($5-$35) development platforms such as the Raspberry Pi has made learning about and adopting the standard more accessible.

Open Source Code and Hardware. To facilitate easy access to this work and attract new developers to the standard, all code and hardware are Open Source. All files are hosted via GitHub® and can be accessed by anyone. The design and complexity of the software architecture abstracts most of the standard functions to simplify integration into other projects. For the hardware, board designs were constrained to use design rules from the most popular PCB prototyping companies.

Verifying the P21451-1 standard. This work not only creates much-needed reference designs on for the standard but also evaluate the standard in the context of the IoT and other paradigms. It was found that while some of the number limitations (for example, only 255 TIMs can be connected to any NCAP Server) seemed outdated, the ideals and services laid out in the document still hold true to designs today. Previously, it was thought that an NCAP Server could only communicate to TIMs and NCAP Clients in only one or two different methods. After investigating the available resources and libraries which accompany platforms such as the RPi and the MSP430, it can be realized that an all-in-one NCAP Server which contains the different methods of communicating to the TIM in one platform. It also can be seen that with the increase in processing power since the creation of the last draft of this standard, that the lines between the NCAP Server and TIM abstraction layers can begin to blur together. This is one area where further research needs to be done, investigating the impact of NCAP Server/TIM hybrids on the performance and autonomy of the network.

Recommendations for Future Work

The future work laid out below pertains mainly to the implementation at Rowan University and the associated research being performed there. One of the most prevalent needs in the implementation is the inclusion of the TEDS services laid out in the IEEE P21451-1 standard. To be able to test these however, work also needs to be done on enhancing the robustness and richness of the TIMs. Either through acquisition or development, to be able to properly test the P21451-1 services, a large number of standard-compliant TIMs are needed. Further work is needed to make a user-friendly SDK or library. This way, a designer can simply add this functionality to their existing projects.

An effort was made to look at the possibility of entirely "virtualizing" the TIM. If another Raspberry Pi, MSP430, or other embedded platforms could contain the same services as a TIM with virtualized sensors and actuators. The work in this type of project can go as far as even allowing designers to simulate single sensor dynamics and even more complex systems such as portions of a factory or buildings. This could be done as an SDK which can then be implemented outside of the embedded platform.

The last recommended work is to expand the Rowan University baseline implementation. This could eventually lead to a campus-wide implementation in the future, giving the Rowan University S.M.A.R.T. lab one of the largest open-source testbeds for IoT devices to date, and could serve as a model Smart City. This would mean additional work such as looking at ArchLinux as an operating system for less power consumption as well as using other methods to connect to the internet such as the Cellular networks. Research into using other messaging protocols such as MQTT, CoAP, and SNMP which are rapidly becoming the defacto standard for IoT development must be considered if this standard and the resources generated are to keep up with the growth of the IoT paradigm.

References

[1] L. Bissi, P. Placidi, A. Scorzoni, I. Elmi and S. Zampolli, "Environmental monitoring system compliant with the IEEE 1451 standard and featuring a simplified transducer interface," *Sensors and Actuators, A: Physical*, vol. 137, no. 1, pp. 175-184, 2007.

[2] M. Choi, K. Cho, J. Hwang and et. al, "Design and implementation of IoT-based HVAC system for future zero energy building," in *2017 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2017*, 2017.

[3] A. Ruano, S. Silva, H. Duarte and P. Ferreria, "Wireless Sensors and IoT Platform for Intelligent HVAC Control," *Applied Sciences*, vol. 8, no. 3, p. 370, 2018.

[4] N. Kularatna and B. H. Sudantha, "An Environmental Air Pollution Monitoring System Based on the IEEE 1451 Standard for Low Cost Requirements," *IEEE Sensors Journal*, vol. 9, no. 4, pp. 415-422, 2008.

[5] F. Zafari and I. Papapanagitou, "Micro-location for Internet of Things equipped Smart Buildings," *IEEE Internet of Things*, vol. 4662, 2015.

[6] Y. Simmhan, P. Ravindra, S. Chatruvedi and M. Hedge, "Towards a data-driven IoT software architecture for smart city utilities: IoT Software Architecture," *Software, practice and experience*, vol. 48, no. 7, pp. 1390-1416, 2018.

[7] V. Paciello, A. Pietrosanto and P. Sommella, "Smart Sensors for Demand Response," *IEEE Sensors Journal*, vol. 17, no. 23, pp. 7611-7620, 2017.

[8] J.-D. Kim, J.-H. Lee, Y.-K. Ham, C.-H. Hong, B.-W. Min and S.-G. Lee, "Sensor-Ball system based on IEEE 1451 for monitoring the condition of power transmission lines," *Sensors and Acuators, A: Physical*, vol. 154, no. 1, pp. 157-168, 2009.

[9] A. Dastjerdi and R. Buyya, "Fog Computing: Helping the Internet of Things Realize Its Potential," *IEEE Computers*, vol. 49, no. 8, pp. 112-116, 2016.

[10] A. Sulyman, S. Oteafy and H. Hassanein, "Expanding the Cellular-IoT Umbrella: An Architectural Approach," *IEEE Wireless Communications*, vol. 24, no. 3, pp. 66-71, 2017.

[11] N. Kouzayha, M. Jaber and Z. Dawy, "Measurement-Based Signaling Management Startegies for Cellular IoT," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1434-1444, 2017.

[12] M. A. Fernandes, S. G. Matos, E. Peres, C. R. Cunha, J. A. Lopez, P. Ferreira, M. Reis and R. Morais, "A framework for wireless sensor networks management for precision viticulture and agriculture based on IEEE 1451 standard," *Computers and Electronics in Agriculture,* vol. 95, pp. 19-30, 2013.

[13] F. Barrero, J. A. Guevara, E. Vargas, S. Toral and M. Vargas, "Networked transducers in intelligent transportation systems based on the IEEE 1451 standard," *Elsevier Computer Standards & Interfaces,* vol. 36, no. 2, pp. 300-311, 2014.

[14] R. Rudman and N. Sexton, "The Internet of Things," HighBeam Research, 1 June 2016. [Online]. Available: https://www.highbeam.com/doc/1P3-4081174771.html. [Accessed 22 June 2017].

[15] M. Kranz, "Why Industry Needs to Accelerate IoT Standards," *IEEE Internet of Things Magazine,* vol. 1, no. 1, pp. 14-18, 2018.

[16] S. Vivek, D. Verma and P. Krishnan, "Towards Solving the IoT Standards Gap," in *2019 International Conference on Advances in Computing, Communications, and Informatics*, 2018.

[17] J. Schmalzel, F. Figueroa, J. Morris and et al., "An Architecture for Intelligent Systems Based on Smart Sensors," *IEEE Transactions on Instrumentation and Measurement,* vol. 54, no. 4, pp. 1612-1616, 2005.

[18] E. Song and K. Lee, "Understanding IEEE 1451 - Networked Smart Transducer Interface Standard," *IEEE Instrumentation and Measurement Magazine,* vol. 11, no. 2, pp. 11-17, 2008.

[19] J. Higuera and J. Polo, "IEEE 1451 Standard in 6LoWPAN Sensor Networks Using a Compact Physical-Layer Transducer Electronic Datasheet," *IEEE Transactions on Instrumentation and Measurement,* vol. 60, no. 8, pp. 2751-2758, 2011.

[20] J. Wei, N. Zhang, N. Wang, D. Lenhert, M. Nielsen and M. Mizuno, "Use of the “smart transducer” concept and IEEE 1451 standards in system integration for precision agriculture," *Computers and Electronics in Agriculture,* vol. 48, no. 3, pp. 245-255, 2005.

[21] R. Wall and A. Huska, "Design Platform for Plug-and-Play IEEE 1451 Traffic Signal," in *IECON Proceedings*, 2005.

[22] E. Song and K. Lee, "STWS: A unified web service for IEEE 1451 smart transducers," *IEEE Transcations on Instrumentation and Measurement*, vol. 57, no. 8, pp. 1749-1756, 2008.

[23] D. Wobschall, "An Implementation of IEEE 1451 NCAP for Internet Access of Serial Port-Based Sensors," in *Sensors for Industry Conference*, Houston, Texas, 2002.

[24] A. Fatecha, J. Guevara and E. Vargas, "Reconfigurable Architecture for Smart Sensor Node Based on IEEE 1451 Standard," *IEEE Sensors*, 2013.

[25] Y. Ma, A. Cherian and D. Wobschall, "A combined ISO/IEC/IEEE 21451-4 and -2 data acquisition module," in *2017 IEEE Sensors Applications Symposium*, Glassboro, NJ, 2017.

[26] W. Kim, S. Lim, J. Ahn, J. Nah and N. Kim, "Integration of IEEE 1451 and HL7 Exchanging Information for Patients' Sensor Data," *Journal of Medical Systems,* vol. 34, no. 6, pp. 1033-1041, 2010.

[27] B. Croitoru, A. Tulbure, M. Abrudean and et al., "Creating a transducer electronic datasheet using I2C serial EEPROM memory and PIC32-based microcontroller development board," *Advanced Topics in Optoelectronics, Microelectronics, and Nanotechnologies VII,* vol. 9258, no. February 2015, pp. 92580W1-92580W9, 2015.

[28] D. Hernández-Rojas, T. Fernández-Caramés, P. Fraga-Lamas and et al., "A plug-and-play human-centered virtual TEDS architecture for the web of things," *Sensors* , vol. 18, no. 7, pp. 1-40, 2018.

[29] D. Zhu, "IOT and big data based cooperative logistical delivery scheduling method and cloud robot system," *Future Generation Computer Systems,* vol. 86, pp. 709-715, 2018.

[30] T. H. Kim, C. Ramos and S. Mohammed, "Smart City and IoT," *Future Generation Computer Systems,* pp. 159-162, 2017.

Appendix A - The “Hello World” metric

The “Hello World” metric is a measure of how many lines of code are needed to print “Hello World” onto the screen. For this test, the common programming languages of C, C++, Java, and Python were compared. This test was not meant to compare these languages in terms of execution time, amount of memory required to utilize, etc., but rather how simple it is for someone with very little background in developing embedded systems to get started with the language. In this metric, the lower the score, the easier it is to work with the programming language.

C is a compiled language, meaning that it is broken down into machine language (a language consisting of fundamental instruction which the processor on the computer can understand) and executed. Because of this, programmers need to direct the compiler to use a library of code which is built into the operating system to allow us to access the terminal or screen on the computer. A library of code is a collection of functions and custom datatypes which can be utilized in other programs. C (and consequently C++) also require a “main” function which is the epicenter of the entire program. This is the function which will execute upon starting the program after any initialization is required. Because of all this, the resulting code to print “Hello World” on the screen is:

```
#include

int main(void)
{
    puts("Hello, world!");
}
```

Not included above are the required Linux commands to compile this code into an executable file which the computer can then run. Without counting whitespaces or empty lines, C receives a Hello World metric score of 5 Significant Lines of Code (SLOCs), when the braces containing the function are considered.

C++, which was built off of C and is compiled, retains many of the necessities of C to print “Hello World” as can be seen below. Due to the addition of the need for the return, C++ receives a Hello World metric score of 6 SLOCs.

```
#include

int main()
{
    std::cout << "Hello, world!";
    return 0;
}
```

Java is built around the idea of object-oriented code and it lends itself greatly to applications dealing with data or datasets. Java is not just a compiled language in the way C or C++ is. It is compiled down into bytecode which ten requires a Java Virtual Machine to interpret it. Much like C++ and C, the program needs to reach into the operating system and call upon a set of functions and drivers to print to the screen within a terminal window. This can be seen below:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.printIn(“Hello World”);
    }
}
```

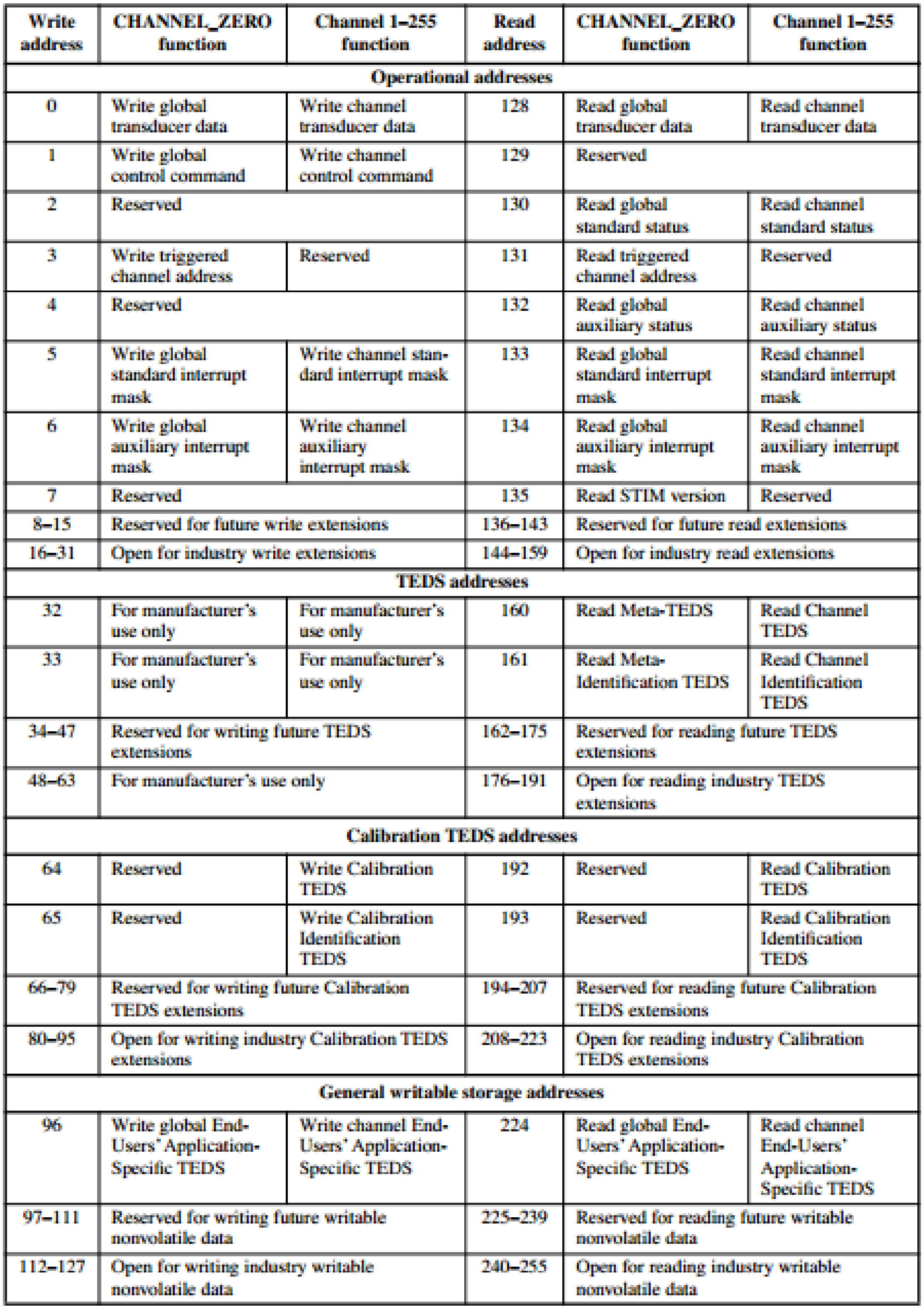
Including the terminating braces, Java is at a tie with C with a Hello World Metric Score of 5 SLOCs.

Unlike the previous examples, Python has built-in functions which encapsulate a lot of the complexities into high-level functions, removing a lot of the steepness in the learning curve. Since it is also an interpreted language like Java, it requires (in most cases) a Python runtime environment to execute the code. Python also can be treated as a scripted language where the developer directly tells the program what to do, instead of generating specific functions (such as in all the previous examples), running each line of code in order. Python can be used in the object-oriented ways as seen above, however if it is scripter, “Hello World” can be printed using the simple code below:

```
        print “Hello World”
```

Coming in at the lowest possible Hello World Metric score of 1, the intuitive nature of the Python language, combined with the strong operating system level support within the Raspbian operating system made this the best choice for the main programming language for the NCAP Server. Using this would allow new students on the project, and eventually, developers attempting to work with the standard, a quick climb up the associated learning curve so that they can get to what is important.

Appendix B – Table of TIM Services



Write address	CHANNEL_ZERO function	Channel 1–255 function	Read address	CHANNEL_ZERO function	Channel 1–255 function
Operational addresses					
0	Write global transducer data	Write channel transducer data	128	Read global transducer data	Read channel transducer data
1	Write global control command	Write channel control command	129	Reserved	
2	Reserved		130	Read global standard status	Read channel standard status
3	Write triggered channel address	Reserved	131	Read triggered channel address	Reserved
4	Reserved		132	Read global auxiliary status	Read channel auxiliary status
5	Write global standard interrupt mask	Write channel standard interrupt mask	133	Read global standard interrupt mask	Read channel standard interrupt mask
6	Write global auxiliary interrupt mask	Write channel auxiliary interrupt mask	134	Read global auxiliary interrupt mask	Read channel auxiliary interrupt mask
7	Reserved		135	Read STIM version	Reserved
8–15	Reserved for future write extensions		136–143	Reserved for future read extensions	
16–31	Open for industry write extensions		144–159	Open for industry read extensions	
TEDS addresses					
32	For manufacturer's use only	For manufacturer's use only	160	Read Meta-TEDS	Read Channel TEDS
33	For manufacturer's use only	For manufacturer's use only	161	Read Meta-Identification TEDS	Read Channel Identification TEDS
34–47	Reserved for writing future TEDS extensions		162–175	Reserved for reading future TEDS extensions	
48–63	For manufacturer's use only		176–191	Open for reading industry TEDS extensions	
Calibration TEDS addresses					
64	Reserved	Write Calibration TEDS	192	Reserved	Read Calibration TEDS
65	Reserved	Write Calibration Identification TEDS	193	Reserved	Read Calibration Identification TEDS
66–79	Reserved for writing future Calibration TEDS extensions		194–207	Reserved for reading future Calibration TEDS extensions	
80–95	Open for writing industry Calibration TEDS extensions		208–223	Open for reading industry Calibration TEDS extensions	
General writable storage addresses					
96	Write global End-Users' Application-Specific TEDS	Write channel End-Users' Application-Specific TEDS	224	Read global End-Users' Application-Specific TEDS	Read channel End-Users' Application-Specific TEDS
97–111	Reserved for writing future writable nonvolatile data		225–239	Reserved for reading future writable nonvolatile data	
112–127	Open for writing industry writable nonvolatile data		240–255	Open for reading industry writable nonvolatile data	