# WEB SERVICE ARCHITECTURE

# FRAMEWORK FOR EMBEDDED DEVICES

By

Paul David Yanzick

MS in Computer Science, 2005
BS in Computer Science, 2004

A Dissertation Submitted to the Faculty of Colorado Technical University in Partial Fulfillment of the Requirements for the Degree of Doctor of Computer Science

Colorado Springs, Colorado

June 2009

WEB SERVICE ARCHITECTURE

FRAMEWORK FOR EMBEDDED DEVICES

By

Paul Yanzick

Approved:

[illegible]

Tim Maifeld, Ph.D.
Professor of Computer Science
(Director of Dissertation)

[illegible]

Phil Burian, Ph.D
Professor of Computer Science
(Committee Member)

[illegible]

Cynthia Calongne, DCS
Professor of Computer Science
(Committee Member)

7/7/2009

Date Approved

## ABSTRACT

The use of Service Oriented Architectures, namely web services, has become a widely adopted method for transfer of data between systems across the Internet as well as the Enterprise. Adopting a similar approach to embedded devices is also starting to emerge as personal devices and sensor networks are becoming more common in the industry. This research examines web service encoding mechanisms currently in place and their system resource utilization on both a full computer system and an embedded device. Research was conducted to first baseline SOAP and JSON serialization and deserialization performance, and then develop a web service architecture for an embedded device and verify similar performance characteristics exist. In the process of developing the second experiment, a flexible Python-based web service architecture is developed and implemented on an embedded device. Performance of SOAP, XML over HTTP, and JSON encoding mechanisms are measured and compared in this environment. Attempts to implement a similarly functioning architecture using a Java framework called XINS failed due to resource constraints imposed by the embedded device.

In every case tested, JSON encoding significantly outperformed the tested XML-based encoding, showing clear advantages in both processing performance and network bandwidth utilization. Also shown is that XML over HTTP has performance characteristics similar to that of JSON for smaller, less complicated data objects, but JSON is significantly more efficient when more complicated data objects are used.

## DEDICATION

This research is dedicated to my wife Monica and my son Jayden. Without your encouragement, support, and sacrifice, I would not have completed this research.

Also, this research is dedicated to my parents, David and Dorothy, for without their support to pursue my interests and dreams I would not have made it this far. I wish you were here to see this day.

## ACKNOWLEDGMENTS

I'd like to thank my committee, Dr. Maifeld, Dr. Burian, and Dr. Calongne for all of their support and guidance throughout this process. Without their support I would never have made it through this process. It is people like these that make the Doctoral program at Colorado Technical University so valuable to its students.

I would also like to thank the Engineering folks at Sencore for their guidance regarding the attachment of sensors and probes to an embedded device without letting the 'magic smoke' from the microcontroller and associated chips. Without their guidance, I would have likely needed to buy several new embedded devices throughout this process.

Finally, but most importantly, I thank my family for their support and encouragement. My wife Monica and my son Jayden have sacrificed a lot for me to dedicate time for this endeavor, and without their support I would not have been in this program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

# CHAPTER I

# INTRODUCTION

## 1.1 Topic Introduction

With the introduction and subsequent expansion of the World Wide Web, an explosion of software development paradigms and technologies have been developed and applied to other aspects of the science of computers. Most of these technologies are focused in providing a mechanism for the traditional application to migrate to a web-enabled application allowing decentralized and scalable availability to a variety of devices and users. The Service Oriented paradigm encompasses this set of technologies, specifically in the use of web services.

The use of services for web applications has become very popular in creating web environments that interact with the user much like a traditional application would, constantly pushing the envelope of software development. Also challenging, a push for fault resilient software solutions have become ever more requested by software consumers. Web services or the use of other Service Oriented Architectures (SOA) have come to the forefront as a method for creating this type of interaction and availability across varying platforms. According to Thomas Erl, the top selling SOA author, creator and editor of *SOA Magazine*, and contributor to the SOA community through published articles and research, the following is a definition for a Service Oriented Architecture [1]:

> "SOA establishes an architectural model that aims to enhance the efficiency, agility, and productivity of an enterprise by positioning services as the primary

means through which solution logic is represented in support of the realization of strategic goals associated with service-oriented computing."

The use of SOA is not strictly reserved for web-based interaction. These technologies can and have been adapted to other arenas, such as in applications where a universal and standardized protocol is required. Sensor networks are also implementing web service architectures to standardize communication.

Many industrial devices [2, 3] make use of embedded controllers and technologies to keep the size and power consumption down. Vendors of such devices, in order to meet these stringent requirements, create devices that are specific to their purpose and have just enough system resources to perform their job. In doing so, proprietary protocols and communication busses are developed to streamline the performance of their particular device.

Sensor networks are made up of potentially thousands of devices known as motes that have relatively limited resources for processing, sometimes even fewer than industrial control devices depending on their use. Each device serves a specific purpose, such as relaying temperature data or video to a central location. These devices integrate together to create a network that is capable of anything from monitoring industrial processes or manufacturing plant operations to use in military surveillance systems, and beyond. Devices used in a sensor network are designed with low power consumption in mind, since they are usually not connected to a power source other than an on-board battery. Depending on the implementation, sensor networks (particularly wireless sensor networks) are capable of adapting to their environment and adjusting communication methods to accommodate their new environment.

While there are similarities between sensor network motes and industrial control devices, they typically serve different purposes. For example, an industrial control device usually has external sensors attached to it that allow for it to perform its function. In addition, they perform a certain level of logic on the data, commonly using ladder logic, such that they can interact with their environment. Sensor network motes, on the other hand, generally do not have that type of interaction. Generally motes of similar size and resource configuration are used for data acquisition only.

Despite their differences, industrial control devices and sensor networks suffer from similar problems. For example, network bandwidth is a precious commodity when multiple devices are required to share it. Constant communication can absorb all available bandwidth causing network collisions and lost communications. These issues can severely limit the expandability of such solutions to a point that they may not be useful. Updating the network technology used can increase performance, but it is not the most economical solution and does not solve the problem long-term.

## 1.2 Problem Definition

The interoperability of sensor networks, industrial control devices, and other proprietary embedded device solutions such as mobile communication devices, smart phones specifically, can be improved by using a universal communications mechanism. Such a mechanism can then be used across a network environment allowing data to be transferred in a consistent manner. The ability for web services to function in a cross-platform environment using well-known protocols is a good fit for such a solution.

The resource utilization of the protocols, however, is of concern. Web service protocols such as SOAP were not designed for the use in an embedded environment, however it does provide a universal mechanism for the transmission of data. SOAP is XML-based, which can quickly consume available system resources. Other protocols exist that perform the same function, which are not XML-based. It is assumed that replacing SOAP with a different protocol that does not use XML, more efficient system resource utilization could be realized.

## 1.3 Hypothesis

The hypothesis for this research is the following:

**Ha:** Web services can reduce system resource utilization by using a transmission protocol that is not XML-based.

The associated null hypothesis is the following:

**$H_0$:** Web services using an XML-based protocol would see the same or higher system resource utilization by using a transmission protocol that is not XML-based.

As a clarification, the term *system resource utilization* represents system resource use such as CPU time and network bandwidth use.

## 1.4 Research Questions

The most important question to answer is whether or not web services can achieve better resource utilization, and through that higher performance, using a different protocol than SOAP as its transport mechanism on an embedded device. In order to determine if this is achievable, it must be divided into more specific questions.

Can a web service protocol be found that provides higher performance than SOAP? Can that performance difference bet attributed to XML in some way? Critical to this question is identifying the protocols that are currently used by web service architectures and determine their performance characteristics in regard to processing time and network utilization requirements. Testing these key metrics can identify which protocol is better performing.

Once the characteristics of the protocols have been found, do these performance patterns map directly to use on an embedded device? The performance on an embedded device can vary as resources are constrained significantly in comparison with a web service provided by a company web server. Performing similar measurements on an embedded device will provide the required information to determine if the performance model identified can be applied to an embedded device.

Is the identified performance increase associated with better resource utilization? The utilization of system resources can be attributed to a higher performing solution, but not in all cases. Some solutions perform consistently regardless of the number of resources made available to them. Other potential performance latency issues can also limit the performance of the solution, identified by the protocols functioning at the same performance level.

## 1.5 Proof Criteria

This section examines the null hypothesis and provides a set of proof criteria. These criteria are broken into two separate sections: criteria that identify an end point for the research work and criteria that identify success, rejecting the null hypothesis.

As identified above, the null hypothesis is as follows:

**$H_0$:** Web services using an XML-based protocol would see the same or higher system resource utilization by using a transmission protocol that is not XML-based.

To simplify this statement, changing the protocol used in a web service architecture to one that is not XML-based will not increase the performance of the system and could possibly decrease performance.

To signify the completion of the work, the following proof criteria will be used:

- The selected encoding methods from both experiments show a trend indicating that methods not using XML require less processing time for each data object type.
- The selected encoding methods from both experiments show that methods not using XML require less network bandwidth for each data object type.
- The selected encoding methods implemented on an embedded device, which do not use XML, indicate higher performance per character input.

The following criteria are used to determine that the null hypothesis has been rejected.

- For the selected protocols, a protocol that is not XML-based has lower system resource utilization in both an embedded environment as well as a system environment in which resources are not constrained.

## 1.6 Repeatability

Since the purpose of this research is to find variations of the mean system resource utilization in an interval, the sample must truly contain the mean of the population data. For this type of research, a complete population cannot be obtained since an almost infinite number of parameters can affect the experiment. Due to this fact, a representative sample of the population distribution will be gathered using a large number of runs in each experiment with as many parameters fixed as possible. For instance, each protocol and data type combination will be executed 1000 times.

## 1.7 Assumptions

Several assumptions are made to focus the scope of this research. Some have already been identified above, however are repeated below for consistency. They are not listed in any specific order.

All communication between devices will take place over a wired network. Wireless networking can introduce variables that will affect the performance of communications, and by removing this component these potential performance issues are also removed.

Network communication on the wired network will not contain any network devices such as switches or routers. Communication between devices will take place through an isolated crossover cable, thus removing any degradation of communication performance caused by the network devices themselves. While this does not remove performance latency during the process of communication completely, it does remove components that are not required and will reduce the latency of data transfer focusing on the web services themselves.

All functions required for web service communication will not necessarily implemented in all experiments. The goal of this research is to isolate the system resource utilization of the transmission protocol specifically, not additional functions required for full web service functionality.

The embedded device used for testing will be a resource constricted device, but not to the point that it restricts the ability to test the resource utilization of various web service frameworks that may exist. The device must be flexible enough to be reconfigured while at the same time have resource restrictions to better emulate an embedded device that may be used in a sensor network or as an industrial control device.

## 1.8 Dependencies

As a core dependency for this research, a set of protocols or encoding mechanisms that are used for web service communication must be identified. Several protocols must be selected so they can be compared to identify a protocol that has less system resource utilization than an XML-based communication protocol.

Some risk is involved when selecting ways to implement these protocols. Coding the protocols from scratch can introduce inconsistencies and additional performance bottlenecks when compared to existing implementations. Using existing implementations or frameworks are susceptible to the same issues.

The data sets that are used may not show a significant use of system resources when applied to the selected set of protocols. Data sets that are small may perform equally well in any of the selected protocols.

Finally, results may not be easily reproducible using a different embedded device or environment. A different device may utilize a different operating environment, which can affect the system resource utilization significantly. In addition, the hardware configuration of the embedded device can be significant in reproduction of results. For example, having less memory on the device can cause additional memory swapping or cause the web service to cease functioning altogether.

## 1.9 Limitations

While embedded devices are readily available for experimentation, finding one that provides the flexibility needed for this research was challenging. Having multiple embedded devices with varying hardware configurations available to perform an analysis of utilized resources would allow for a more complete investigation, however device availability and funding constraints limited the quantity of devices that could be purchased. The resources a particular embedded device has available can create different resource utilization analysis results when compared to another device. For this reason, only one embedded device model was selected for this purpose, proving the concept.

The number of possible data sets that could be evaluated is almost limitless. The selected data sets were limited to nine, representing a simple message being passed containing a single value to a complex structure that is multiple layers deep. With the selected data sets, performance utilization characteristics can be captured which will identify anything from the simplest of communications to a complex object environment.

## 1.10 Constraints

To properly demonstrate that other protocols not XML-based have lower system resource utilization than their XML-based counterparts, a subset of available protocols needs to be selected. For this reason, protocols that are commonly used at the time this research was developed were used. The intention was not to create a new protocol, but to identify those that have lower performance utilization and can be used on an embedded device.

While this research focuses on Service Oriented Architectures and embedded devices, an embedded device that is flexible enough to support multiple configurations was selected. Many devices exist that can be used as the base platform for this research [4, 5], however due to the time required to work with each, using a device that can easily be reconfigured will allow for more focus on the research than the hardware implementations that could be used.

# CHAPTER II

# LITERATURE REVIEW

## 2.1 Introduction

A significant amount of research has been completed in the use of SOA in varying environments, as well as the protocols that are used for communications. This chapter outlines the research that has been completed surrounding sensor network architectures as well as SOA and its commonly used protocols. Performance characteristics are also examined in the reviews.

## 2.2 Design Methodologies

In a paper written by Bonfatti, Gadda, and Monari [6], an approach for supporting the initial phases of the Programmable Logic Controller (PLC) software lifecycle were explored. The aim was to support the critical phases such as requirement specification, requirement analysis, and software design. Through their research, they developed a method called EASIER, which draws on the object- oriented paradigm. A case study was performed using what they called the 'picking bay,' a component of a large automated warehouse. The system is designed to select specific goods and remove them from the warehouse. Their results indicated that hard problems could be solved using the object-oriented model.

Service Oriented Architecture (SOA) has been incorporated into sensor networks with the research of individuals from the University of Prestoria [7]. The authors discuss the challenges that have been created by wireless sensor networks as well as actor networks. The authors implemented a WSAN solution named Cerberus, a home security system, and documented the difficulties they had with the solution as well as offering a proposed solution called OSGi. OSGi implements SOA technologies for the management of the network infrastructure. One of the major issues that the authors cite as being a problem is the ability to develop software applications in such an environment. Another issue that they identified is the fact that it is difficult to expand existing systems features. The authors cited an example of adding health monitoring to a system that does not already have that ability.

Bonivento, involved in the OSGi solution, performed further investigation into sensor network, specifically for industrial applications[8]. A broad methodology is presented discussing a process to map an application to a sensor network. The process, at a level, starts with a description of the control algorithm to be used as well as possible hardware for the solution. A case study of a control system for use in a manufacturing environment is included, showing how the methodology covers the complete design process.

The use of sensor networks is not limited to use in industrial applications. Common objects, such as a vehicle, also contain such networks. Several individuals from Daimler-Benz and Mercedes-Benz [9] presented activities within their organizations that were focused on developing an optimal process for designing electronic control units (ECUs) which are used in automotive applications. The authors indicate several examples where such sensors are used, such as vehicle drive train and braking systems.

Three levels of the development process are identified, as well as methods and tools that are to be used with such as process.

Web services can be used as a communication method for sensor networks, as Van Engelen [10] has shown. Code generation techniques used to implement XML web services on embedded devices are discussed. A product called gSOAP [11] is used for implementation of a SOAP web service, utilizing the C and C++ programming languages. Van Engelen describes the methodologies behind SOAP communication and how gSOAP implements them. An example is provided which gathers the temperature of a sensor probe and sends it via the web service. Several gSOAP optimizations are discussed which are included in the latest release of the product.

In this paper [12], the authors use gSOAP to build a toolkit for web services called Devices Profile for Web Services (DPWS). The goal of DPWS is to allow resource-constrained devices to host web services in their entirety. Discovery, description, subscription, and event handling is incorporated within this solution. The SOAP encoding method is used in this solution for messaging between devices. The authors believe DPWS has promise in automated industrial environments since it is targeted specifically for resource-constrained devices.

The discussion of a web service layer, called middleware, is discussed in this paper [13] as a possible solution for reducing the complication of developing a distributed solution using network-capable embedded systems. The authors identify how middleware has not been introduced in the design flow of these solutions, and how middleware can support various development paradigms. An implementation of this system has been completed using a wireless sensor network. Three paradigms (Database, Tuplespace, and Object-Oriented) were tested and analyzed in the paper.

The authors of this paper [14] discuss the creation of a new language that can be used to make the development process for sensor network applications easier for developers. A framework was created called Sensor Network Application Construction Kit (SNACK) that acts as a set of 'building blocks' allowing an application to be developed quickly and easily. A simple data acquisition application was developed using the newly formulated framework.

A framework called Rappit [15] is discussed in this paper, identifying the shortcomings of high level scripting when used with embedded devices. The Rappit framework is designed to alleviate some of the shortcomings due to abstraction such as speed of execution and a shortened development cycles. The framework was successfully used in the development of a mini-FDPM system.

In this paper [16], the authors discuss the challenges of developing wireless sensor network solutions, such as power usage and resource constraints. They discuss combining a low-level protocol stack with management capabilities, implementing the stack so it is not so resource intensive.

Requirements for the development of sensor network applications is evaluated in this paper [17]. The authors have identified one of the key elements as being an overall system architecture. The requirements for such a solution were discussed and a small device was developed as well as an operating system called TinyOS.

The authors of this paper [18] discuss the implementation of the directed diffusion, data centric paradigm for coordination of various nodes on a sensor network. Every node on the network is aware of the application, and allows energy savings in the way that paths are selected and data is cached as it is sent across the network. The authors study this technique using a remote-surveillance sensor network.

Depending on the sensor network type, it may need to operate in a real-time environment. This paper by M. N. Matelan [19] discusses a model for the development of a real-time control device which consists of three main areas needing to be considered during the design process. The first area is the selection of the hardware. The second, the development of the software to run on the selected hardware platform, and the final area is the monitoring of the system to verify real-time integrity. The author introduces the concept of hardware binding, and indicates that binding the software to the hardware later in the design cycle helps develop a machine-independent solution.

In this paper [20], the authors believe that emerging technologies will see to the end of the traditional computer. Instead, systems will facilitate everyday living. Due to the number of devices that are available, a framework is needed to allow all of them as well as other actors to communicate together developing an intelligent system. An approach for a distributed web-based management framework is discussed, called DAFNY. The OSGi framework is also referenced in this paper.

The management of a large distributed network can be a very challenging problem. The authors of this paper [21] indicate that ‘current solutions are a combination of centralized management and significant over-provisioning of the infrastructure.’ A service for monitoring such a solution, $S^3$, is examined. This solution is designed to detect failures on the network, adaptive resource placement, and higher performance.

The sensors on a sensor network, also known as motes, function with limited resources at their disposal as they frequently function on battery power. Most utilize a battery or a solar power solution, making them susceptible to power loss causing them to fall offline. When this happens, the remaining sensors on the network may need to be re-programmed to take on additional roles. Doing so consumes additional power from the remaining nodes, as it is not a very power efficient operation. The others of this paper [22] discuss the use of a virtual machine infrastructure to assist with the process of re-provisioning the motes. Using a virtual machine, complex programs can be made to be very small, requiring less power to transmit and configure.

## 2.3 Implementations

A number of sensor network implementations exist in industry, as it is a versatile technology with a lot of capability. The following implementations provide examples of how such a grouping of embedded devices can be used, as well as problems that can occur with the use of them.

A particular sensor network implementation was used in the development of an Emergency Medical Service (EMS), which allowed communication of patient status to hospitals while they are being transported in an ambulance [23]. An overview of existing practices is discussed in the paper, as well as the architecture of the system. As a proof of concept, an ambulance was driven throughout a city containing two 'patients.' It was found that the transmission of patient information was delayed, possibly caused by the implementation of the web service architectures that were used. Noted were overhead of the SOAP protocol and the parsing of XML information. They also note that "binary data gets expanded (by an average of 5-fold) when included in XML, and also requires encoding/decoding of this data," and since the implementation utilizes wireless technologies with less bandwidth for data transport, it increases the network congestion introducing additional delay.

Sensor network systems are not only used in private industry, but in military systems as well. VigilNet [24] is a sensor network designed for surveillance of opponents in the field of battle. It is often dangerous for human personnel to attempt to perform surveillance activities; however being able to deploy unmanned surveillance missions will eliminate the need to put individuals in danger. The paper further discussed the development of the VigilNet system and an evaluation of its middleware components that are the core foundation for the system.

The idea for distributed network devices has been around for a number of years. In this paper [25] mini-computer devices are discussed in reference to the automation of a laboratory. The goal of the project was to reduce the amount of time that was required to perform six months worth of experiments to a timeframe of two weeks.

## 2.4 Service Oriented Architectures

Service Oriented Architectures are becoming more prevalent as design architectures for modern solutions due to their ability to be scalable and re-usable. This technology can also be used in sensor networks and industrial controls to relay information across the shared network.

In this paper [26], the authors discuss the fact that the World Wide Web has become successful due to the design of its software architecture, and its ability to service as a distributed hypermedia application. The modern architecture focuses on scalability, generalization independent and intermediary components to increase performance and communicate with legacy technologies. A technology that is currently used in SOA is Representational State Transfer (REST) is introduced in the paper, and compared to other current web architectures.

REST is also discussed in this paper [27], stating it "guided the recreation and expansion of the modern web." The authors believe that some of the capabilities of REST are underutilized, and discuss additional web architectures. During their process, an expansion of REST, Computational REST (CREST) is introduced.

As with any technology, there are always difficulties with implementation. In this paper [28], the authors discuss the fact that SOA architectures involve 'the use of incomplete information.' For example, it is possible that a web service may not be available at a time that it is required. This can be a challenge for the web service consumer, as it will require more information to switch to the new or alternate solution. The authors state that they are performing research to developing a process for addressing these issues based on the use of semantic web services.

Various individuals and groups have attempted to analyze the trends of technology developed and used on the World Wide Web. Mehdi Jazayeri has done so in this paper[29], stating that since the dawn of the Internet, the applications that use it have become more complex in nature. He explains that the Web has expanded in function from providing static content to becoming a platform for applications. He further discussed the fact that web development technologies have adopted techniques from the Software Engineering field and that he believes future advances in browser technologies will advance the Web even further.

SOA is not made up of a specific technology, but is a composite of varying technologies that work together to form the solution. Some of the technologies used in AJAX, a framework designed for partial post back of web pages, contain technologies that SOAs utilize. This paper discusses a framework called Exhibit [30] which utilizes JSON, a lightweight language used to describe JavaScript objects, for presenting content on a web page. The author indicates that large companies and organizations have exploited technologies allowing them to provide richer content than what can be provided by an individual, and that the use of this technology will help put the individual on a level playing field.

More specific to sensor networks and embedded devices, this paper [31] discusses the benefits of using SOA for communication between devices that are resource-constrained smart devices, as well as provides a discussion on such an implementation. A method for integration of 'dumb' devices into such a solution using a gateway device is also discussed in the paper.

## 2.5 Protocol Performance Characteristics

Service Oriented Architectures can make use of a number of protocols for communication purposes, however the popular encoding method uses the SOAP protocol.

This section describes some of the research that has been completed regarding web service performance, specifically regarding the protocols used.

This paper [32] analyzes the performance of SOAP for the use of mobile applications. The authors found that SOAP used over HTTP and TCP is not efficient and can create a high amount of latency during transmission due to the overhead that SOAP creates as well as in the transmission protocol TCP. The utilization of UDP was found to provide a higher throughput compared to HTTP and TCP for the transmission of SOAP.

The SOAP protocol utilizes an XML format to form the basic structure of the protocol. Since the majority of currently implemented web service architectures utilize the SOAP transmission protocol, they must be able to process XML to encode and decode the data transmitted via the web service. The authors of this paper [33] develop a method for benchmarking XML parsers and used their testing method on several common XML parsing solutions. Inefficiencies were found in these solutions that would affect their performance, and led the development of a high performance parser implementation based on their research.

XML itself has advantages as well as disadvantages that it brings to the SOAP protocol as Schmelzer [34] has identified. XML uses a highly structured format and is human readable, allowing the data being transmitted within it to be accessible without the use of special tools for decoding. Another key advantage is that it is very flexible, allowing other languages to be defined using it, such as the SOAP protocol. These benefits come at a steep price however as XML requires significant processing power to encode and decode the data stored within. Allowing the data to be human readable does not allow for binary data to be stored easily. And finally, the amount of space required to represent data in XML can be very large due to the structure of an XML document.

Lawrence [35] agrees that while XML is easily readable as it is self describing and is a standardized format for data transfer, it is not space efficient. In an experiment that he had performed utilizing varying formats for storing data, he had calculated the overhead required to store data in a particular format, per attribute. His findings are shown below in Table 1.

**Table 1 – Overhead Per Attribute For File Formats [35]**

| Format | Overhead Per Attribute |
|---|---|
| CSV | 1 (or 3) |
| FSV | S – D |
| XML (encode empty) | 2*N+5 |
| XML (no empty) | (2*N+5)*(1-E) |

A comma separated values file (CSV) is the only format that Lawrence investigated that has constant overhead per attribute recorded. FSV files are proportional to the size of data being stored. The overhead is the total size allowed for the data item to be stored with the actual data size subtracted. XML is listed twice in the table as XML can contain null values but still contain the XML markup representing the attribute. In the case that null attributes do not have tags stored, the overhead is twice the size of the attribute name plus five characters for tag delimiting characters. If delimiters with null values are stored in an XML file, the probability that the field has data is multiplied by the size of the attribute name and delimiting characters.

Lawrence further identifies that alternatives to XML have been identified, such as JavaScript Object Notation (JSON) [36] which utilizes fewer delimiter characters at the expense of some human readability. Figure 1 provides an example of a JSON response producing a random integer value with the attribute name of 'simpleInteger'.

```
{"result": {"simpleInteger": 9039}}
```

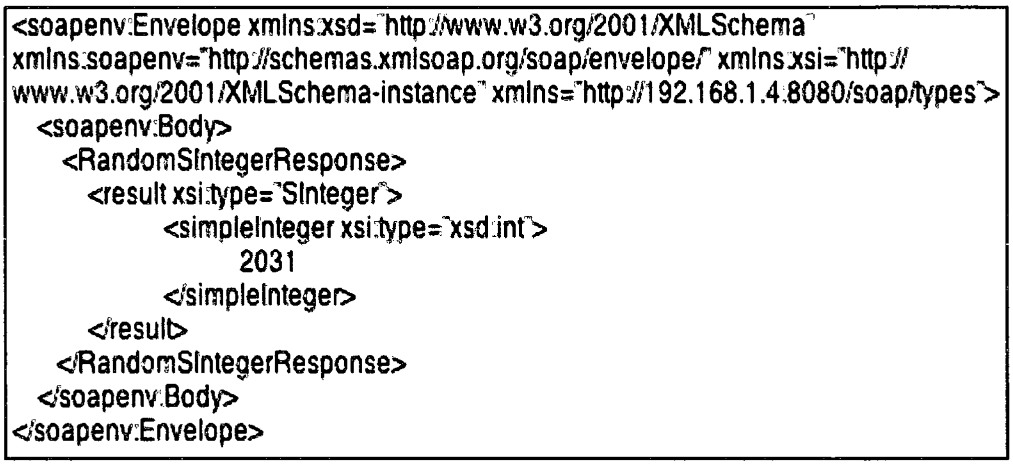
**Figure 1 - Example JSON Response**

Attempts have been made at improving the performance of the SOAP protocol by using a different method for encoding XML. Ng identifies a table driven version of XML (TDXML) [37] that is more efficient as it provides messages that are more compact in a simpler structure. Ng found that TDXML can be incorporated into a SOAP implementation, and has shown that there can be an improvement in performance issues caused by the latency of XML by as much as 30%. If TDXML is used in place of SOAP, performance increases of over 100% can be achieved.

## 2.6 Web Services

Part of the overall Service Oriented Architecture paradigm, web services have been a common method for communication among systems across the web. As with other SOA, web services use a variety of technologies to achieve their ultimate goal.

The authors of this paper [38] provide details of the components that make a web service, with XML being used as the basis. Simple Object Access Protocol [39] is the most common protocol used in a web services architecture, which is designed to be a lightweight XML-based protocol used to transfer data between a client and a server. An advantage of SOAP is that it is truly a platform independent protocol, and it is human readable. Figure 2 shows a sample SOAP response from a web service storing a randomly generated integer.



```
<soapenv:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xmlns="http://192.168.1.4:8080/soap/types">
  <soapenv:Body>
    <RandomSIntegerResponse>
      <result xsi:type="SInteger">
            <simpleInteger xsi:type="xsd:int">
                  2031
            </simpleInteger>
      </result>
    </RandomSIntegerResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

**Figure 2 - Example SOAP Response**

Part of the overall Service Oriented Architecture paradigm, web services have been a common method for communication among systems across the web. As with other SOA, web services use a variety of technologies to achieve their ultimate goal.

Web services use an XML-based language for describing the capabilities of the service as well as what the service requires for input for it to process the request correctly. The protocol that is used for transport, endpoints of the web service, as well as a list of all of the available services at a particular endpoint are all provided by a Web Services Description Language (WSDL) [40]file, typically hosted by the web service endpoint. Figure 3 shows a segment of the WSDL file that defines the web services that produced the SOAP response in Figure 2. Note that the segment identifies two separate methods. The first method, RandomSInteger, requires no parameters to be sent, and has no return. RandomSIntegerResponse provides a result of type SInteger, which in this example is a class object containing a single integer value.

```
- <xsd:element name="RandomSInteger">
  - <xsd:complexType>
      <xsd:sequence> </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
- <xsd:element name="RandomSIntegerResponse">
  - <xsd:complexType>
    - <xsd:sequence>
        <xsd:element name="result" type="types:SInteger"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

**Figure 3 - Example WSDL Function Definition**

The final component that the paper [38] identifies as a component of a web service is the Universal Description, Discovery, and Integration (UDDI) [41] which allow a web service to be discovered. It is a SOAP-based application interface that allows for the publishing and discovery of web services.

Using a web service architecture on an embedded device is the topic of a paper written by Schall, Aiello, and Dustdar [42]. Mobile computing is quickly becoming more popular and the authors identify issues between operating systems and hardware platforms used as a basis for smart phones and other such devices. The use of web services can provide the loosely coupled architecture required for such an environment.

In their paper [42], several toolkits were exampled. gSOAP [10, 11], being a C++ implementation of a web services architecture was compared with several Java toolkits. Several metrics were examined in the study, such as the average time required to execute a request, the latency given for a number of requests to be performed, and the Java overhead required to support the Java toolkit implementations. The authors found that the C++ implementation was quicker at creating the request for data, but slower than the Java toolkits that were examined in providing a response. Overall, however, the C++ implementations outperformed the Java toolkit implementations.

# CHAPTER III

# METHODOLOGY

This chapter describes the plan that was adopted for this research in terms of the work that needed to be completed as well as the approach that was taken. Details of hardware selection criteria and devices that were examined as potential development platforms are also explored in this chapter.

## 3.1 New Contributions

This research contributes to the body of knowledge by implementing a Service Oriented Architecture [1] on an embedded device [5] using a flexible open source toolset [43, 44] with the goal of measuring performance of varying encoding mechanisms. Until now, little research has been conducted on the combination of embedded devices with a Service Oriented Architecture implementation, specifically with a focus on data encoding performance. The resulting framework opens the door to intelligent devices that can be used in the industrial control and sensor network arenas.

The research that was conducted applies web service encoding methods to an embedded device for the purpose of measuring performance characteristics of the device as well as the network that is attached. Most prior research compared the performance of existing protocols on devices that are not resource constrained, such as a desktop or server system [45-47]. Utilization of these protocols on a resource-constrained device can show different performance characteristics.

Another aspect of this research is that it focuses on using objects of varying sizes and depths containing randomly generated data, unbiased by using any specific data. An additional aspect is the characteristics that were captured during the experimentation process which include the following measurements:

- Size of the data transferred across the network using both the Microsoft .Net Framework and the embedded platform
- Total time to encode and decode an object using the .Net platform
- Total amount of time required for the embedded device to process and return a response

These measurements allow for an analysis of how varying platforms perform using web service protocols, as well as a comparison between a desktop computer and an embedded device implementation.

## 3.2 Prior Research

A significant amount of background investigation was conducted to capture the body of research relating to web services in general, how they are applied to embedded devices such as sensor network nodes, varying protocols and protocol implementations that exist, as well as high level performance characteristics for several of the commonly used protocols. In addition, references have been collected for existing web service frameworks and architectures, including specific implementations of such solutions and their uses.

Little research [12] was found that has been focused on web service protocol system resource utilization or performance considerations of these protocols on embedded devices. This research is meant to bridge this gap that exists in the current body of knowledge.

## 3.3 Data Structure Selection

A set of sample data structures or objects was required for experimentation that could be easily generated by the web service architecture of increasing complexity. Simple objects containing a single layer and variable to objects that were multiple layers deep were required to best determine how each protocol would perform and cover the gamut of potential data structure complexities that a solution may require. A total of nine data structures were chosen that contain integer and/or text values to varying degrees of complexity. Composite data structures make up several of the measured data structures. Details of the actual data sets used are described in **Appendix A**.

## 3.4 Experimental Approach

Determining the performance between several web service protocols does not lend itself to an analytical approach. Experiments were conducted on an embedded platform in order to accurately determine their performance in such an environment. Virtualized environments can be created rather than using actual hardware for this purpose, however virtualization can cause performance variation between hardware due to the algorithm and pattern choices made during its development. Results can be difficult to recreate between different virtualization environments for the same reasons.

Two separate experiments were conducted to determine the performance of web service protocols. The first experiment was used to determine the performance of the SOAP protocol, commonly used in web services, as well as other protocols that can be used for the same purpose. The second experiment determined the performance of these protocols on an embedded device, validating if web service architectures can be utilized on such a platform.

## 3.5 Protocol Selection

Selection of the web service protocols is a key component in this research. The protocols that were selected are protocols that provide the element of human readability. This, by definition, removes any binary implementations of web service communication protocols, even though they are able to transfer data at a much higher density without much overhead.

Creation of a new web service communication protocol is not the intention of this research. The use of protocols that have been established and used in a web service environment increase the probability that a client will not require additional or proprietary software in order to communicate with the new implementation. The key is that a high performing web service protocol is selected that can be used on an embedded device without causing the device to fail due to its constrained resource configuration.

The SOAP protocol [39] was included as it is the most commonly used web service protocol at the creation of this research. It is also XML-based, allowing for this protocol to be the reference protocol to compare system resource utilization against.

JavaScript Object Notation (JSON) [36] is selected as it is the other end of the spectrum when compared with SOAP. JSON does not use XML and does not have the overhead that SOAP requires.

In between SOAP and JSON is XML [48] over HTTP, which is the final protocol that was examined. Using XML over HTTP does not require the header information that SOAP does, however is verbose in its tag formatting like SOAP.

All three of these protocols are capable of encoding a data structure and transmitting it to the web service client, allowing the analysis of system resource utilization using varying complexities of objects. These protocols are identified in Table 2 below, indicating which methods use an XML format and which do not.

**Table 2 – Web Service Protocols and XML Utilization**

| Protocols Researched | XML Used |
|---|---|
| SOAP | Yes |
| JSON | No |
| XML over HTTP | Yes |

## 3.6 Experiments

As mentioned in earlier sections, multiple experiments were required during the process of completing this research. The intention of the first experiment was to validate that a difference in system resource utilization exists between XML-based and non XML-based web service protocols while the second experiment verified the results found on an embedded device. A summary of these experiments and their purpose is shown in Table 3.

**Table 3 – Summary Of Experiments**

| Experiment Number | Purpose |
|---|---|
| 1 | Proof of concept, Microsoft .Net measurements |
| 2 | Embedded device implementation, measurements |

### *3.6.1 Experiment 1*

To validate that web service protocol resource utilization is meaningful to research, an experiment was conducted to measure the resource usage of both the serialization and deserialization actions on data to be transferred by a web service. Both the web service host and client are required to convert the data that they receive and send to the other. Figure 4 shows the process in which the data being passed through a web service must endure.

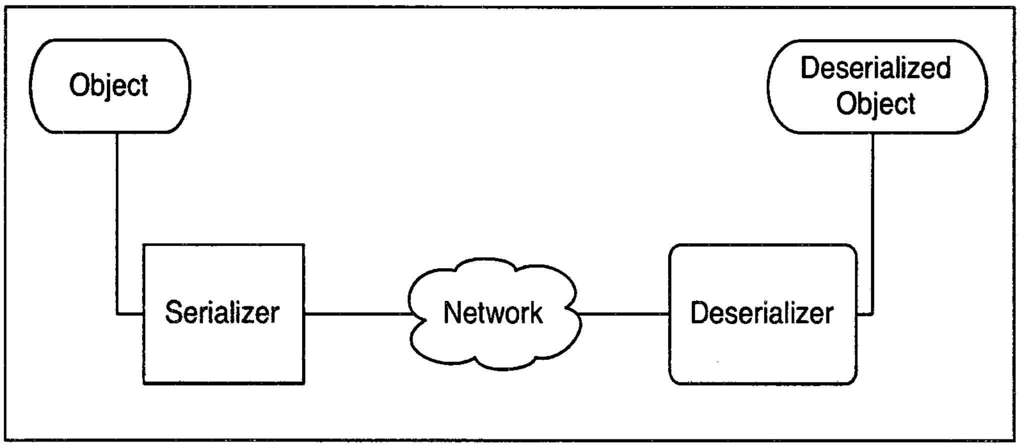


**Figure 4 - Serialization/Deserialization Process**

The data being transferred through a web service may or may not be a data structure. A simple value such as an integer could be transferred in its place, therefore the term 'object' in Figure 4 represents any data that is to be transferred through the web service.

The data must first be serialized into some format, dictated by the web service transfer protocol, so that it may be sent to the device receiving the request. The data is sent in its encoded form across the network medium (local area network, wide area network, Internet) to the receiving device. The data is then decoded back into its original format and processed.

Experiment 1 created an environment in which selected data structures will be populated with random data, serialized, and deserialized. The Microsoft .Net 3.5 Framework was used in a Windows environment for implementation of the experiment as it supports web service connectivity and two web service protocols, SOAP and JSON. Microsoft .Net 3.5 has improved the JSON architecture and supports the methods used in Microsoft .Net 2.0, so both versions of JSON encoding will be utilized. The new implementation of JSON in the Microsoft framework will be referred to as JSON Contract throughout this document.

Both the serialization and deserialization timings were recorded, as well as the total encoded message length. The measurement of data encoding and decoding provides an estimated time value indicating the length of time required for the system to process and convert the data that is sent or received. The character length of the encoded message provided details regarding the size of the network transmission required to send the message to the destination device.

### *3.6.2 Experiment 2*

Determining the resource utilization of each protocol on a complete computer system provides insight into the potential performance on an embedded device, however as the computer system is not as resource constrained, it is not an accurate model. Implementing web services on an embedded device will accurately create an environment in which resource utilization can be captured and then analyzed. The type of embedded device as well as its available resources can vary, which can lead to repeatability issues.

Existing web service frameworks were examined and compared to make a determination as to which would be the best fit for the selected embedded device. The selection of a framework depended on the hardware and operating system configuration of the embedded device. The development language of the framework also played a factor, as the embedded device must be capable of supporting it.

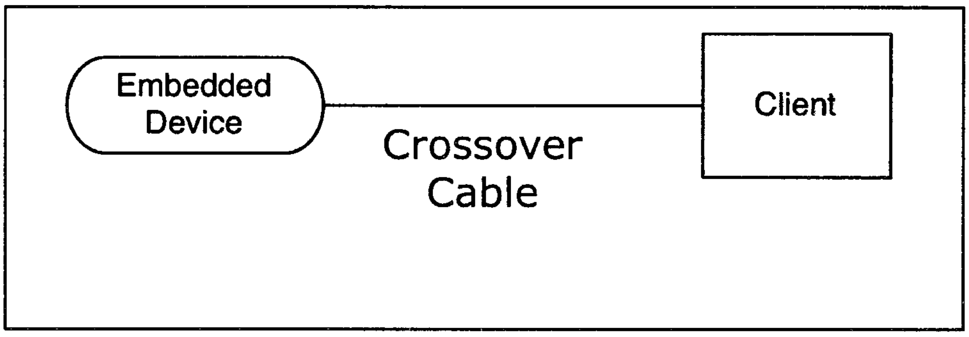


**Figure 5 – Embedded Device and Client Setup**

The embedded device was connected to the client, which will provide a host application that will perform the testing and capture measurements. A crossover cable was used between the embedded device and the client to ensure no outside interference or latency will be introduced into the system. The cable itself was short to reduce signal attenuation. Figure 5 demonstrates a model of the physical implementation of the experiment.

Measurement of transfer time as well as length of data took place on the client. Adding additional burden to a resource-constrained device, however small it may seem, can affect the result set. Reducing the amount of load on the embedded device where it can be performed elsewhere is key to an accurate result set.

The client used the Microsoft .Net 3.5 Framework, however it did not use its built-in web service communication capabilities. The focus of this experiment is to measure the system resource utilization on the embedded device, not the web service client. By creating a method for communicating with the web services hosted on the embedded device instead of using built-in functionality, the communication process can be controlled at a lower level than could be possible otherwise. Information not required by the web service on the embedded can be removed from the web service query in this way. Submitting data using custom HTTP requests allow.

As measurements were taken on the client and not on the embedded device, the processing time the embedded device required to service a request cannot be measured directly. Network latency was calculated by pinging the device with a packet containing approximately the amount of data expected back from the device divided by two. The ping protocol will send back the data that had been sent from the ping originator, therefore the amount of data expected in the reply will have been transferred across the network. Operating system and hardware latency cannot be accounted for, however these will be averaged out through a number of web service queries.

## 3.7 Data Generation

All data used in both experiments was generated randomly and in a consistent fashion. The selected data structures are composed of integer and string primitive types as well as composite objects containing the same. All values generated for a specific data primitive were created using the same algorithm, per experiment, to ensure anomalies will not develop due to algorithm differences. Algorithms change between experiments due to language nuances that prevent the same methodology from being used. The generated values had consistent character length for both experiments; integers were comprised of 4 characters while strings contain 10 characters.

# CHAPTER IV

# EXPERIMENTAL DESIGN

## 4.1 Introduction

Determining the performance difference between web service protocols that execute on a full computer system, and then comparing them with the performance of those protocols on an embedded device can easily be inconsistent depending on what is measured. The best way to compare the performance of the protocols themselves, as well as the performance on each device is by devising two separate experiments. In this way the performance of the protocols can be examined on both platforms and compared to identify performance trends and bottlenecks that are apparent on both platforms.

### *4.1.1 Experiment 1*

To determine the system resource utilization of a web service, a baseline measurement must first take place using a common web service framework. Developing a new framework from scratch could incorporate performance inefficiencies when compared to a fully tested infrastructure. For this reason, the Microsoft .Net framework was used to form the baseline, as it is a common development framework that incorporates web service capabilities using algorithms that have been tested by Microsoft as well as the developers that utilize it daily.

Microsoft .Net 3.5 was chosen for the framework as it has the capabilities of using the SOAP protocol as well as two different formats of JSON. The .Net Framework allows for individual components of the full web services solution to be used. This ability allows for the ability to focus on the protocols themselves.

A common laptop system was used to perform the experiment. The system configuration included 4GB of memory and a dual core Intel 2.60GHz processor running Microsoft Vista Ultimate Edition. The system was freshly loaded for this purposes with services not required for functionality either disabled or stopped.

Using the randomly generated data sets identified previously in this research, the core processing of a web service was emulated on the local system. For each data set type, one thousand objects were created, serialized into the selected protocol, and deserialized. Object size and timing measurements were taken during the process of serialization and deserialization, and then recorded for later analysis.

Since this experiment required web service functionality, a web-based Visual Studio project was used. This allowed for a web GUI to be used to kick off the process for a particular object type using all three available protocols. All data was saved to a file, and calculated before data was rendered on the web page. **Appendix B** provides an example of the 'dashboard' used to launch the experiment for a particular data set type. As shown, a captured representation of one of the objects is shown in its encoded format.

For each object type, the object class being measured is coded and then executed. The interface did not automate the entire experiment, incorporating all data types being analyzed.

#### *4.1.2 Experiment 2*

Once data has been collected and analyzed from Experiment 1forming a baseline, a similar architecture on an embedded device was implemented to compare against. The embedded device does not support Microsoft Windows, so the .Net Framework is not an option. Another solution is required for the functionality of this experiment.

##### *4.1.2.1 Hardware Selection*

Using an embedded device is the best way to emulate an embedded environment, and the selection of the device is paramount to the success of the experiment. The device that is selected must be flexible enough that varying web service architectures can be implemented and tested while at the same time the resources of the device must be constrained to fit the characteristics of an embedded device.

Several companies manufacturer embedded devices, and two devices were selected for examination. Netburner has several offerings that provide the required functionality. The Netburner PK70 [4] was selected as it is resource constrained, yet it is flexible enough in that it can be programmed using C++. The PK70 has a network adapter, plenty of expandability and I/O options, a real-time operating system, and plenty of example code to use. For the purposes of this experiment, however, it does not allow for rapid prototyping. Everything must be developed from scratch in regards web service architecture implementation.

The second device that was examined and eventually selected for this experiment was the Technologic Systems TS-7800 [5]. The TS-7800 embedded device shares required capabilities with the Netburner PK70 without the complication required to develop the whole architecture from scratch. The TS-7800 has a Debian Linux operating system pre-installed allowing for Linux packages to be easily installed and compiled on the device. In addition, it provides support for wireless capabilities, plenty of expandability, and the ability to attach SATA hard drives. Being that it runs Linux, the device has more memory and storage resources than the Netburner PK70, making it more flexible. **Appendix C** shows the top view of the controller board.

### *4.1.2.2 Web Service Frameworks*

The Technologic Systems TS-7800 device, being Linux-based, enables the ability to use existing web service frameworks instead of building one from scratch. For this experiment, this is the preferred method. Instead of implementing a framework that may have performance bottlenecks due to algorithm implementation a well-tested framework can be used instead.

The XINS framework [49] is an open source Java-based web service framework which allows rapid prototyping of a web service architecture. Much of the work is done automatically. Documentation as well as WSDL files are generated automatically during the process of web service architecture implementation. Figure 7 below identifies the components of XINS, their interaction, as well as the components that are automatically generated.

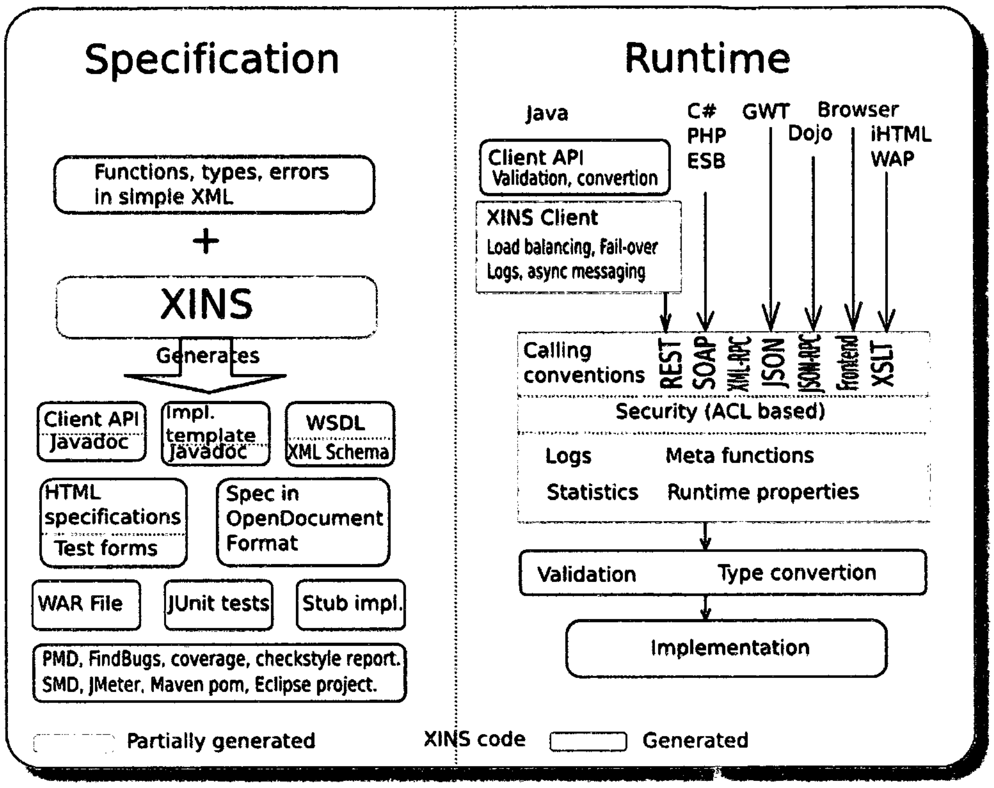


**Figure 6 – XINS Architecture [49]**

XINS uses a series of XML files as well as function definition files [50] to define the web service methods being exposed. Custom types and errors can be defined using the framework allowing for a high level of customizability.

Another web service framework examined is TGWebServices [51] which extends TurboGears [43], Python-based web framework. TGWebServices supports commonly used as well as emerging web service protocols (SOAP, JSON, and XML over HTTP). Like XINS, custom types can be created for use with the web services.

### *4.1.2.3 Web Service Framework Selection*

Both the XINS and TGWebServices frameworks are capable enough to perform the experiment, so both implementations were tried on the TS-7800.

XINS is Java-based, requiring an installation of the Java Development Kit. Implementations of the JDK exist for Linux however the version of Linux pre-installed on the TS-7800, a modified version of Debian 3.1, does not have a pre-built package for its ARM processor type. Building the JDK from source or using an open source implementation of the JDK is possible, however due to the constrained resources of the device or creating an environment outside of the device and porting to the device once ready proved to be a challenge. Updating of the operating system is another option, however the additional space required further limits the available space on the device. Once the JDK was installed and the operating system installed, the available space on the TS-7800 NAND was extremely low with 1MB free. This makes XINS an impractical implementation for an embedded implementation on such a device without additional storage made available or further modification of the core operating system.

TGWebServices and TurboGears both require Python to be installed, which Debian 3.1 does have an implementation of. The operating system on the TS-7800 was upgraded to Debian 4.0 to allow the use of Python 2.5, the recommended version for TurboGears and TGWebServices. The installation of TGWebServices, TurboGears, and its required packages was straight forward using the easy_install command. After installation was completed, the free space on the TS-7800 NAND would allow for further development and applications to be installed.

TGWebServices with TurboGears was selected for this experiment for more reasons than installation difficulty. Since Python does not require compilation like Java, it would be possible to dynamically add web service capability while the application is running. TurboGears will automatically restart itself when it detects a code change, reducing the user overhead in maintaining the infrastructure. Figure 6 below shows the complete architecture of the device. Note that hardware interaction, as demonstrated in Figure 6, can be achieved by writing extension modules to Python.

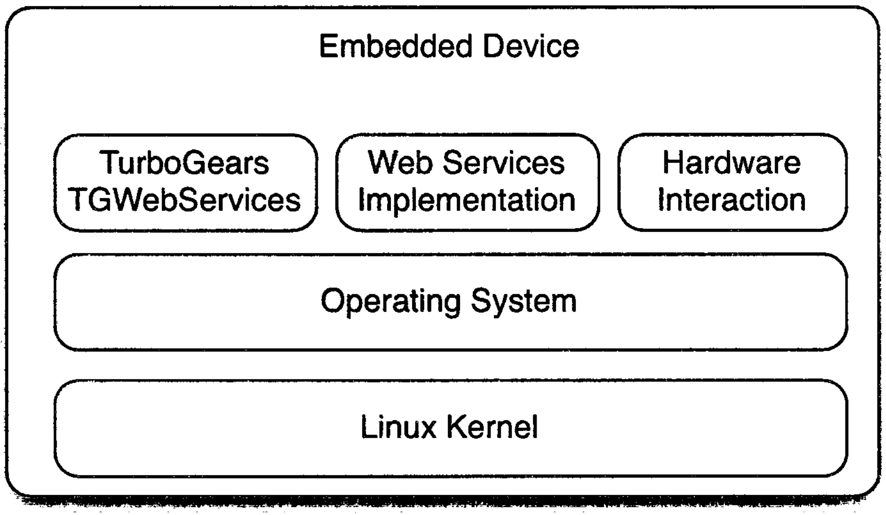


**Figure 7 – Web Service Architecture Block Diagram**

Another significant advantage is that adding custom libraries, providing the ability to perform whatever additional functionality is needed, can easily extend Python. In this way, access to hardware or other low-level functionality is made available for use.

### *4.1.2.4 Experiment Execution*

Due to the nature of the experiment, careful planning was required to ensure that as many variables could be removed as possible. As explained in Chapter 3, an environment was established in which the TS-7800 was directly attached to a computer system. The client system was a laptop running Microsoft Windows Vista Ultimate with 4GB of RAM and an Intel Core 2 Duo 2.0GHz processor. Software applications and services not required for system functionality were stopped or disabled to remove possible bottlenecks from the client system.

The TS-7800 environment was also cleaned in a similar fashion. Default services that were installed from the factory such as MySQL and Apache2 were removed or disabled for the execution of the experiment. No swap was enabled on the embedded device; therefore all execution took place in memory, further emulating a more constrained embedded device.

All data collected is stored on the client system, as well as all measurements that take place to remove additional system resource-consuming services and applications that may adversely affect the embedded device. To perform the measurements, an interface was created to act as a control console to perform experiment execution and data collection.

For each data set type, one thousand objects were requested from the client using an HTTP GET request. The TS-7800 would then create the object, populate it with random data, and send it back to the client. The client would measure the amount of time taken from the time the request was submitted to when the data was received, and subtract the average ping response time to the TS-7800. The ping time was calculated by first querying the TS-7800 to receive an object of which the size is measured. The character length of the returned object divided in half is then sent to the TS-7800 using ping, and the time required for the response to be received is averaged together and subtracted from each request.

The transmission time as well as the character length of the returned object was collected by the client system measure the performance of the protocols used. The data collected is similar to that collected by Experiment 1, allowing a comparison between the two for similar as well as different protocols.

# CHAPTER V

# ANALYSIS OF RESULTS

## 5.1 Validation

Several trial runs were performed during the development of the testing code using varying environments. During the execution, few problems were found with logging of the result data. Using Microsoft Excel, captured data was compared between the experiments and similar trends were identified. The standard deviation from the measurements is small, indicating few outliers exist in the data.

## 5.2 Results

The following sections provide tables and graphs showing the collected data in a graphical format. Because there are a large number of measured values between the two experiments, three groups of data will be shown. The first group of tables and graphs will show the performance timings of the protocols for both Experiment 1 and Experiment 2 separately. The second group will compare the resulting character size for each serialized object for both Experiment 1 and Experiment 2. The graphs in the second group will show little difference between implementations used. The final group will illustrate the true performance of each data type object tested during Experiment 2 by showing the average amount of time required to process each raw data character being encoded.

### *5.2.1 Protocol Analysis*

The data represented in Figure 8 and Figure 9 show the average amount of time required to serialize and deserialize each data structure selected for Experiment 1. The act of serialization takes an object or value and encodes it into a format that is suitable for web services to send the data. Deserialization reverses this process. Note that in all cases SOAP required more CPU time to perform its function than either JSON implementation provided by the Microsoft .Net framework.

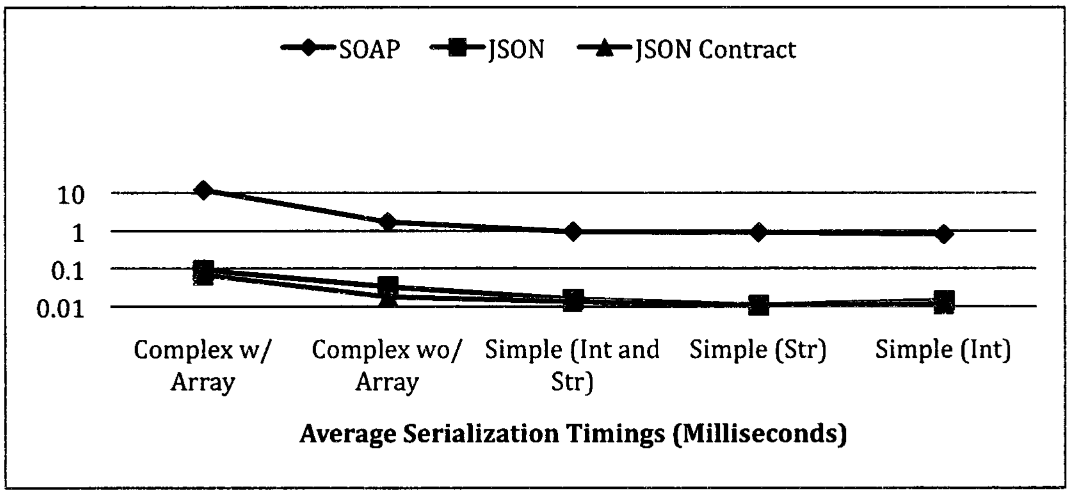


**Figure 8 – Experiment 1 Average Serialization Time (Milliseconds) Per Data Set Type**

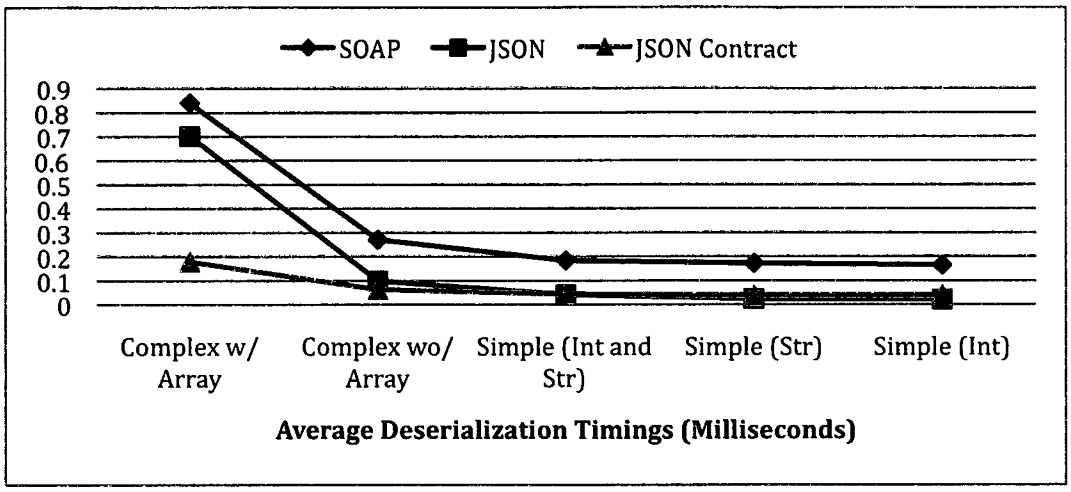


**Figure 9 – Experiment 1 Average Deserialization Time (Milliseconds) Per Data Set Type**

To further illustrate the performance of each encoding method, the following figures compare the time required to serialize and deserialize each data object type using a given encoding method.

Figures 10 and 11 show the average amount of time required to serialize and deserialize the tested data type objects respectively. Note that as the data object becomes more complicated, more time is required to encode or decode. This is to be expected, as more data would generally require more processing time.

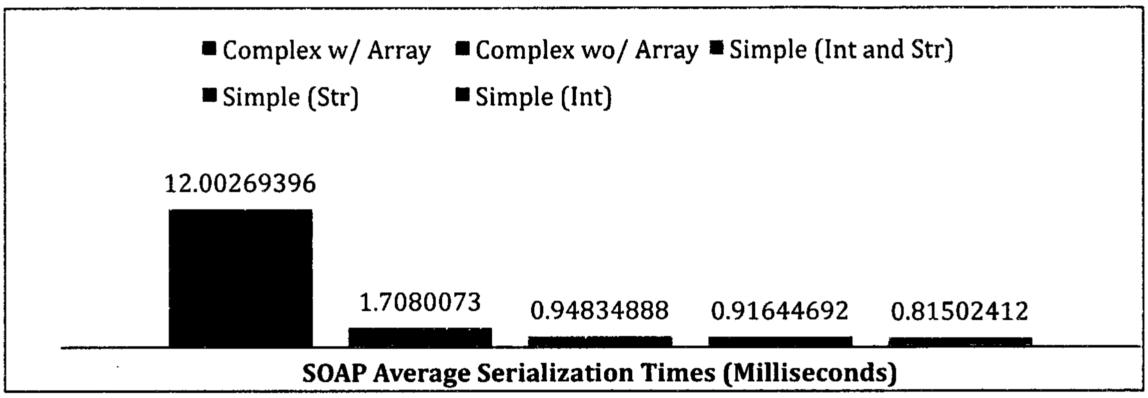


**Figure 10 – Experiment 1 Average SOAP Serialization Time Per Data Set Type**

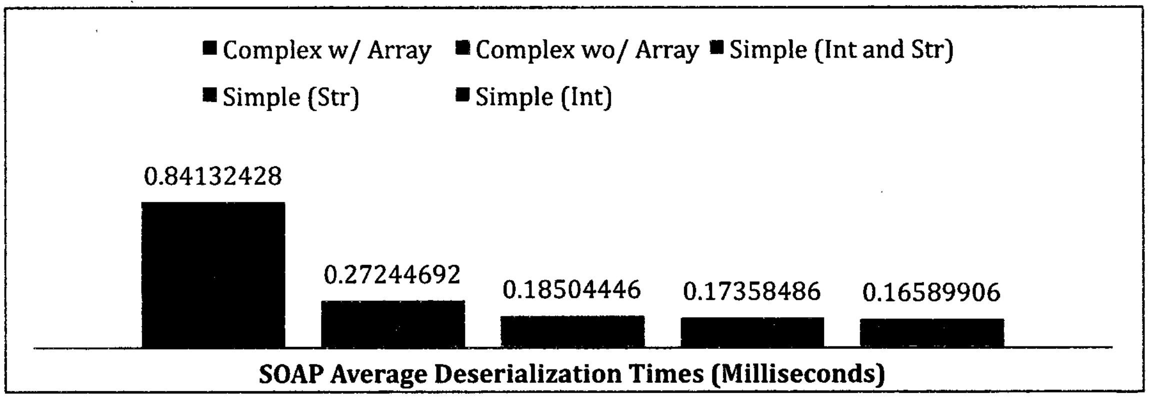


**Figure 11 – Experiment 1 Average SOAP Deserialization Time Per Data Set Type**

The data represented in Figure 12 and Figure 13 show average processor processing time to encode or decode each data object type using the JSON protocol using the implementation introduced in the Microsoft .Net 2.0 Framework. Note that the amount of time to serialize the smallest of the data types is longer than SOAP required. The trend of larger objects requiring additional processing time is otherwise represented.

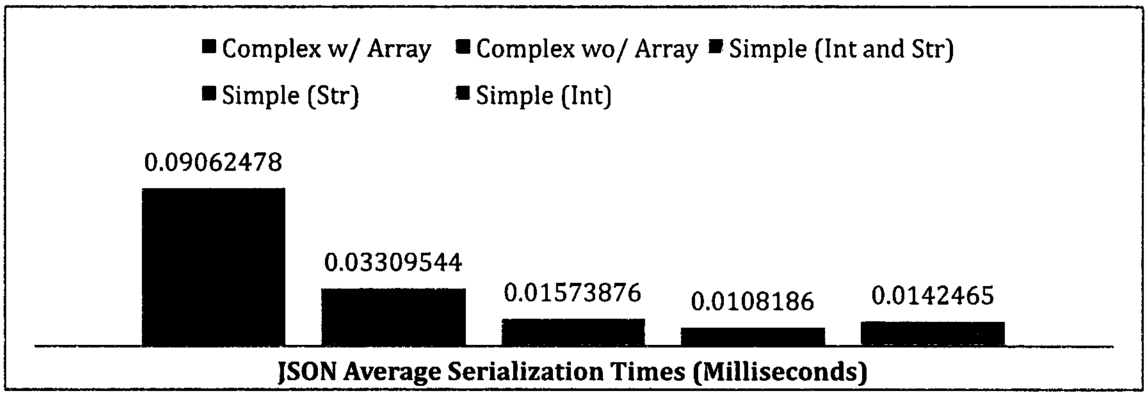


**Figure 12 - Experiment 1 Average JSON Serialization Time Per Data Set Type**

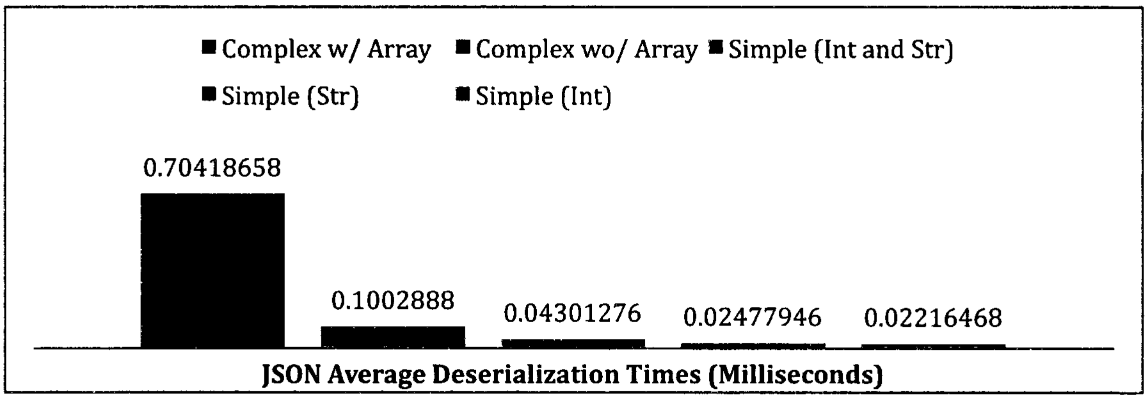


**Figure 13 - Experiment 1 Average JSON Deserialization Time Per Data Set Type**

Figures 14 and 15 provide a comparison of the CPU time required to serialize and deserialize a data object type using JSON Contract, a method of JSON introduced in the Microsoft .Net 3.5 Framework. As with the method of serialization provided by

Microsoft .Net 2.0, the smallest data type object takes slightly more processing time for serialization, however the same is seen in regard to deserialization.

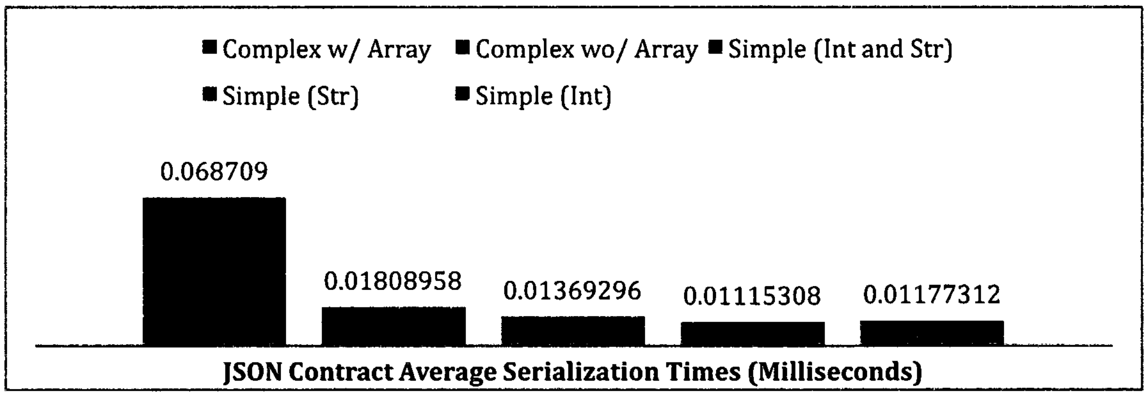


**Figure 14 - Experiment 1 Average JSON Contract Serialization Time Per Data Set Type**

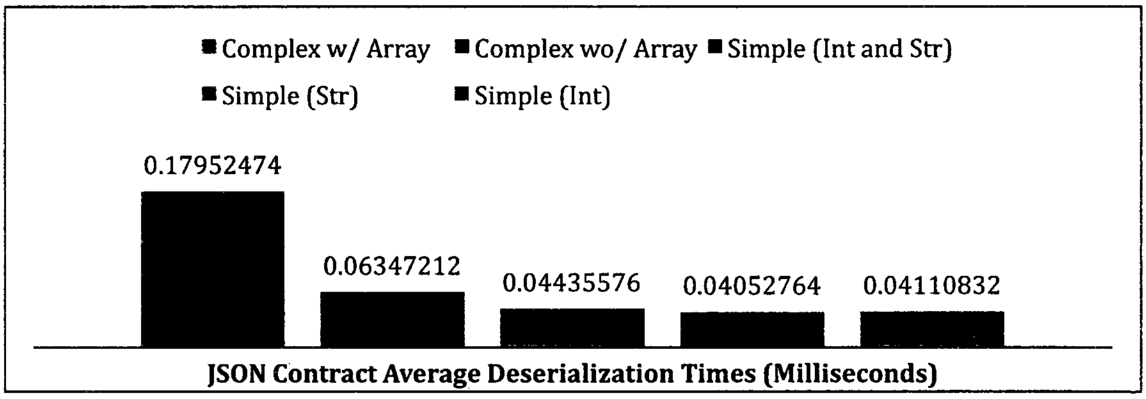


**Figure 15 - Experiment 1 Average JSON Contract Deserialization Time Per Data Set Type**

The data shown in Figures 16 and 17 identify the average time to serialize or deserialize a data object containing an integer primitive type. Note the significantly large delta in processor time used to serialize the object between either version of JSON and

SOAP. The deserialization process does not indicate as large of a difference, however JSON still outperforms SOAP.

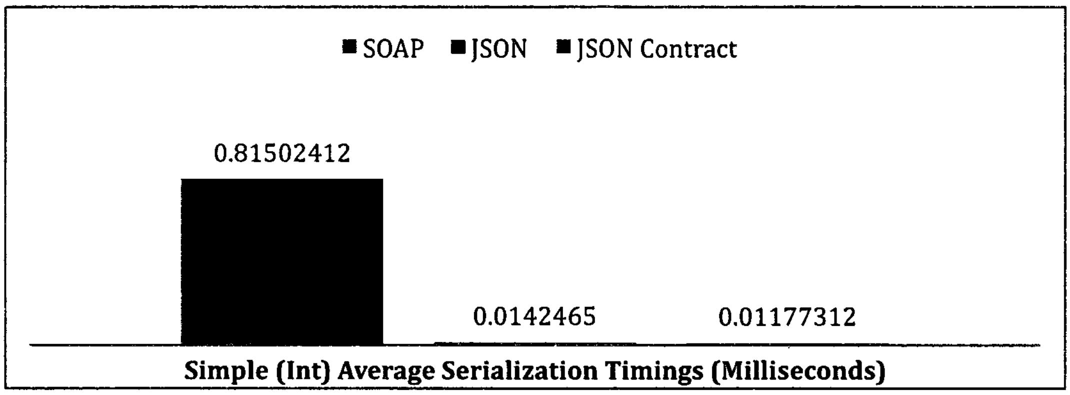


**Figure 16 – Experiment 1 Simple Integer Average Serialization Time Per Protocol**

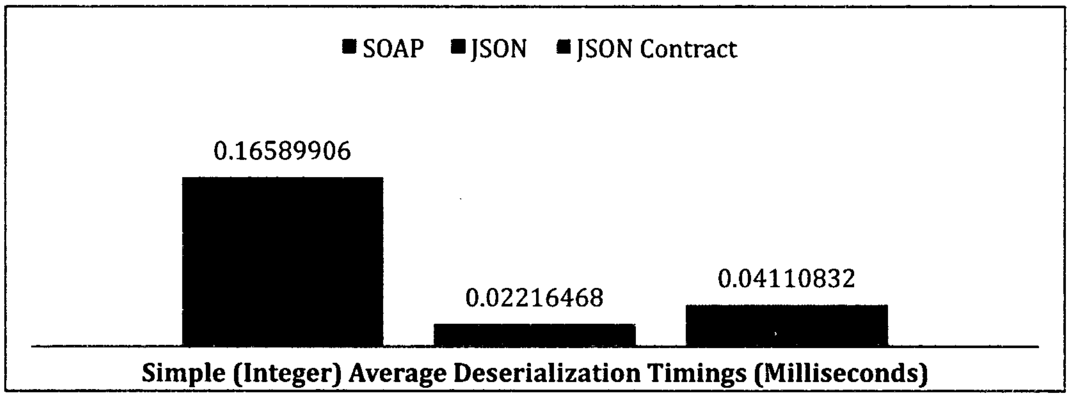


**Figure 17 - Experiment 1 Simple Integer Average Deserialization Time Per Protocol**

A similar trend can be identified in Figures 18 and 19 using a data object containing a string of characters. JSON significantly outperforms SOAP in both the serialization and deserialization processes, however the performance difference is not as significant during the deserialization of the object.

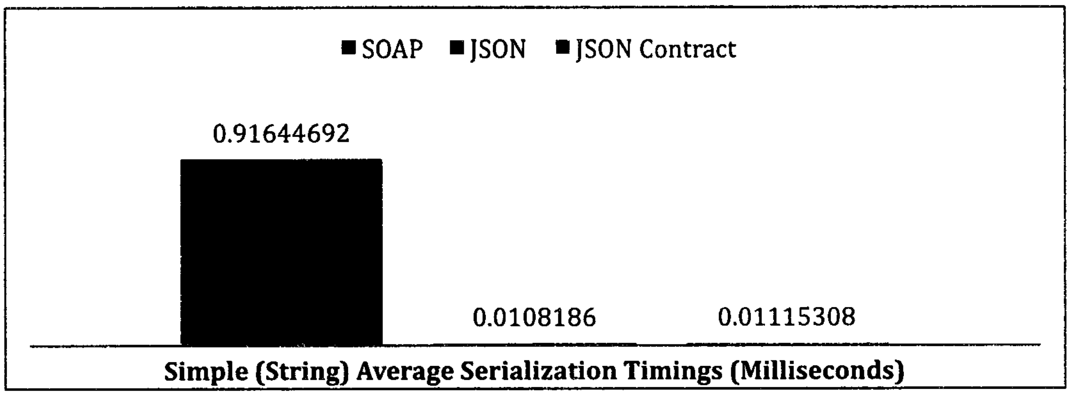


**Figure 18 - Experiment 1 Simple String Average Serialization Time Per Protocol**

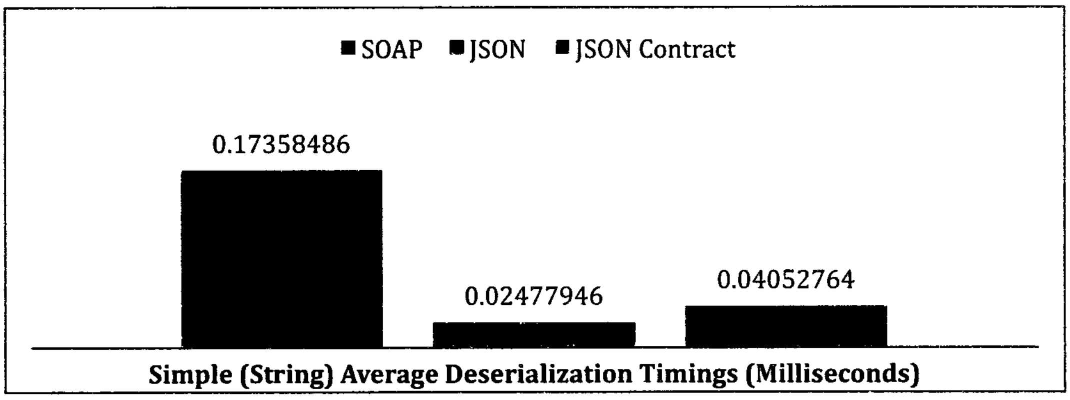


**Figure 19 - Experiment 1 Simple String Average Deserialization Time Per Protocol**

The performance trend can still be identified with data objects containing an integer and string, as shown in Figure 20 and Figure 21. The deserialization process performance difference, while still significant, is decreasing as the object becomes more complicated.

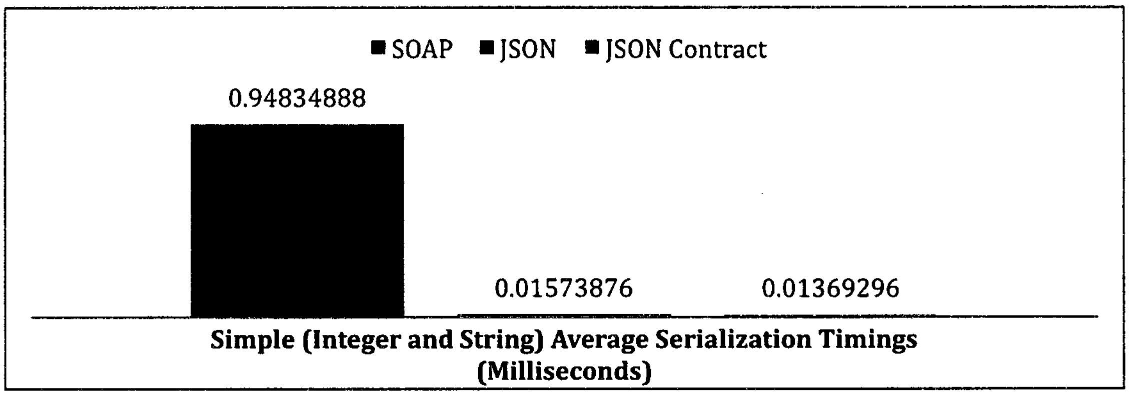


**Figure 20 - Experiment 1 Simple Integer with String Average Serialization Time Per Protocol**

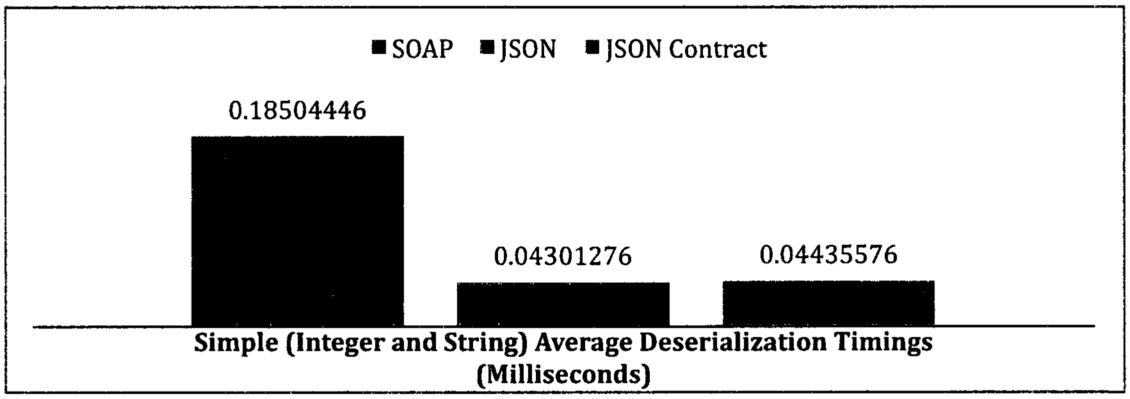


**Figure 21 - Experiment 1 Simple Integer with String Average Deserialization Time Per Protocol**

Figures 22 and 23 show the average time to serialize and deserialize a complex data object containing an integer primitive type, a string value, and a composite object containing an integer primitive and string value. The serialization process still shows a large difference in performance between SOAP and either version of JSON, however the deserialization process indicates a difference between the versions of JSON used.

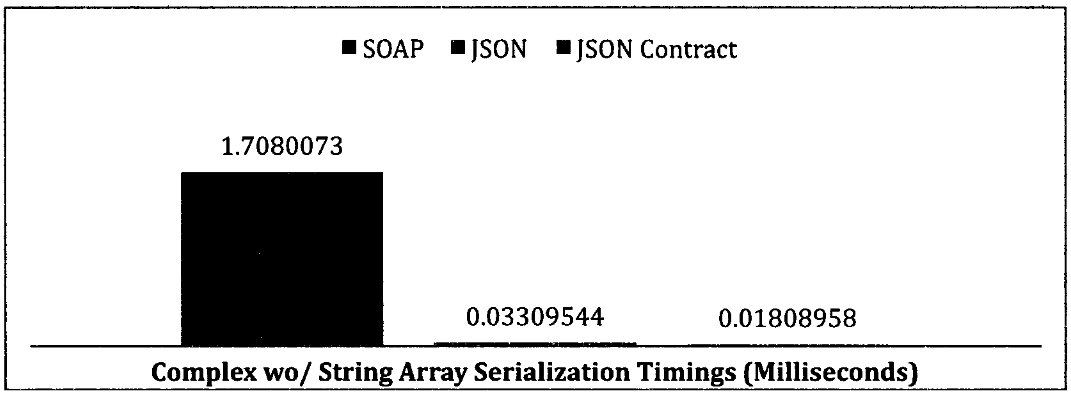


**Figure 22 - Experiment 1 Complex Object without Array Average Serialization Time Per Protocol**

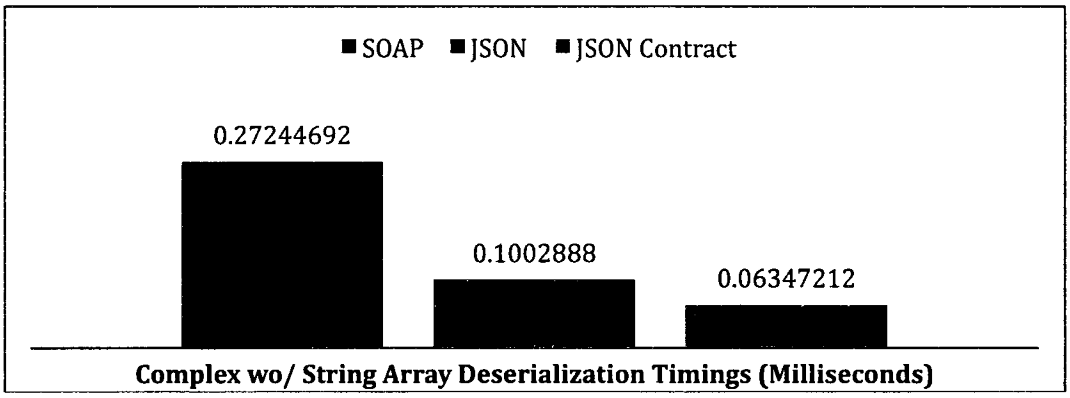


**Figure 23 - Experiment 1 Complex Object without Array Average Deserialization Time Per Protocol**

The performance for serializing and deserializing the most complex data object tested in Experiment 1 is shown in Figures 24 and 25 respectively. The data object measured in this test is made more complex than the object used in the previous test by including a string array containing 50 values. Serialization still shows a significant performance difference, while the deserialization of the object shows a greater delta in performance between the versions of JSON used.

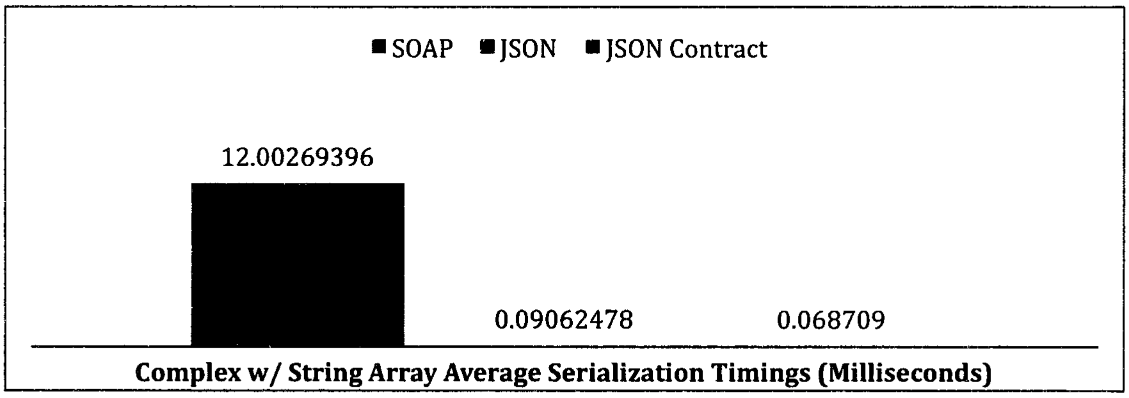


**Figure 24 - Experiment 1 Complex Object with Array Average Serialization Time Per Protocol**

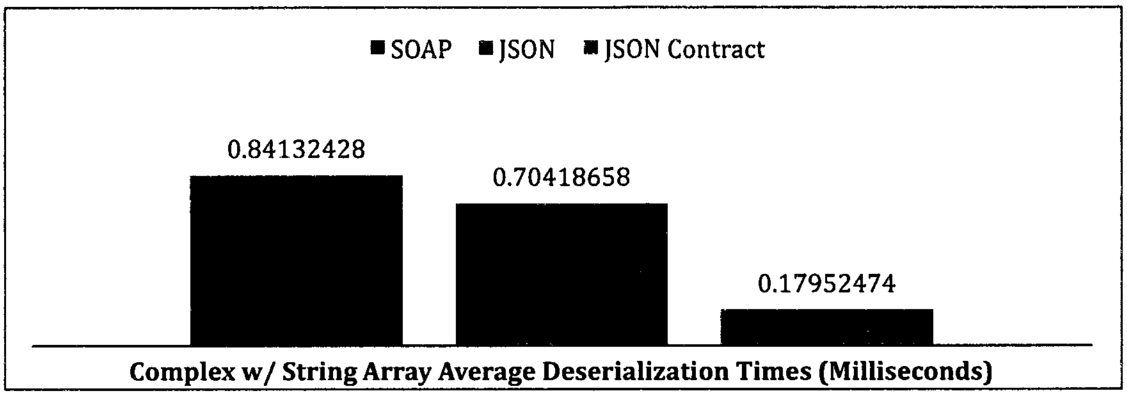


**Figure 25 - Experiment 1 Complex Object with Array Average Deserialization Time Per Protocol**

The following graphs show data collected for the data type objects selected for use in Experiment 2. Note that Experiment 2 utilized TurboGears and the TGWebServices add-on libraries as a base for the web service architecture that was used.

More data than the serialization and deserialization of the data type was collected from Experiment 2. Also captured was the time required for the entire web service to execute to completion, providing a more realistic measurement of the processing required to handle each request.

Figure 26 shows the amount of time required to process a data object type similar to that used in Experiment 1. Note the performance delta between JSON and SOAP does not show as large of a delta. XML over HTTP shows an improvement over SOAP, but appears not as efficient as JSON.

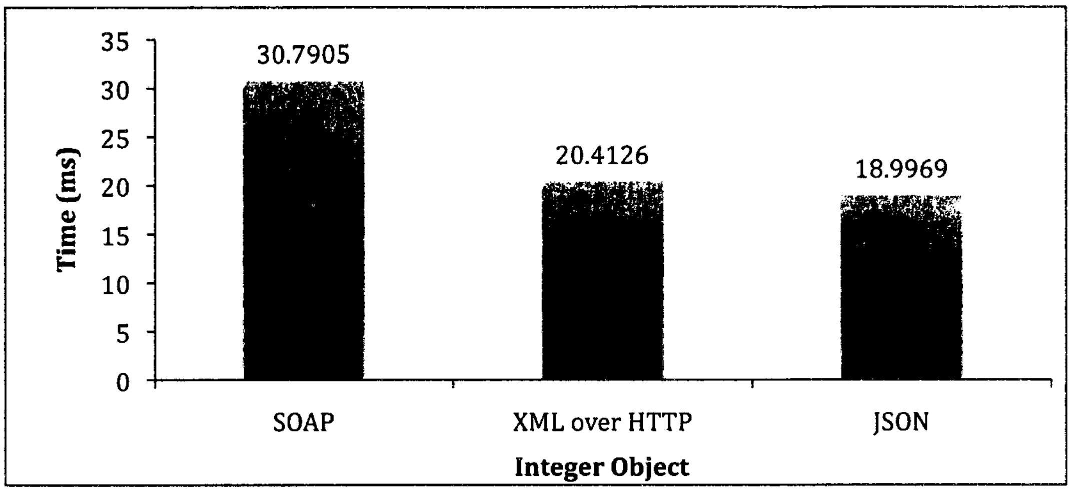


**Figure 26 – Experiment 2 Integer Object Average Round Trip Time**

Figure 27 shows the average processing time for a web service request returning a data object containing a string. Again, XML over HTTP shows an improvement over SOAP, but is close to the performance of JSON.

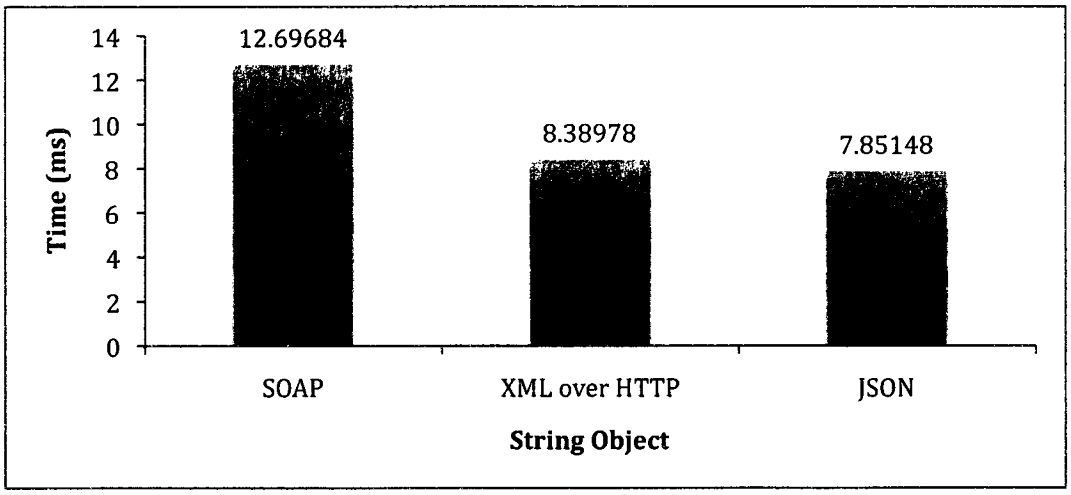


**Figure 27 – Experiment 2 String Object Average Round Trip Time**

A data type object containing both an integer and string was created and tested, similar to that used in Experiment 1. The average time of processing is shown in Figure 28. The same trend is shown as with previous objects, however the time required to process an object containing only a string is larger than processing an object with a string and an integer value.

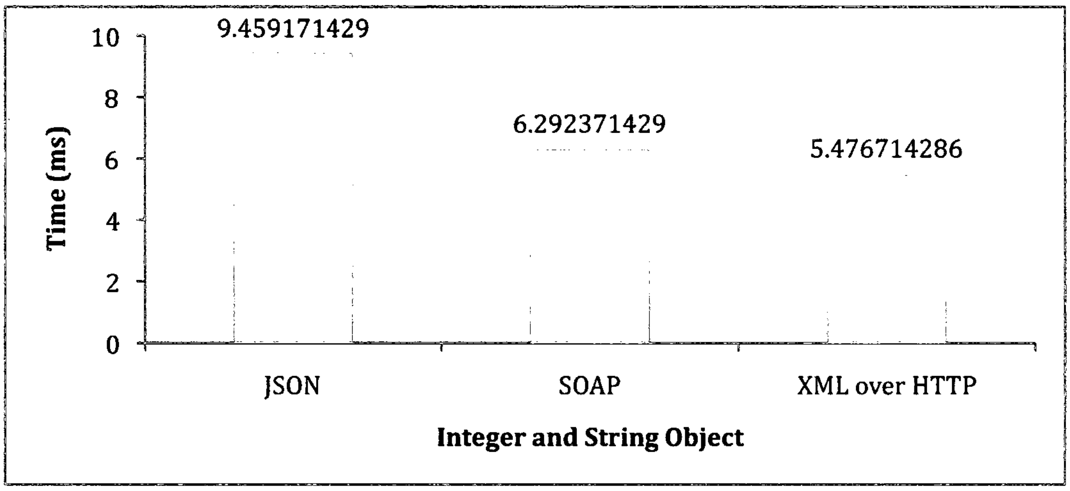


**Figure 28 – Experiment 2 Integer and String Object Average Round Trip Time**

Figure 29 identifies the average processing time for processing a complex data type object, similar in design as the data type object used in Experiment 1. The same trend can be identified with this object.

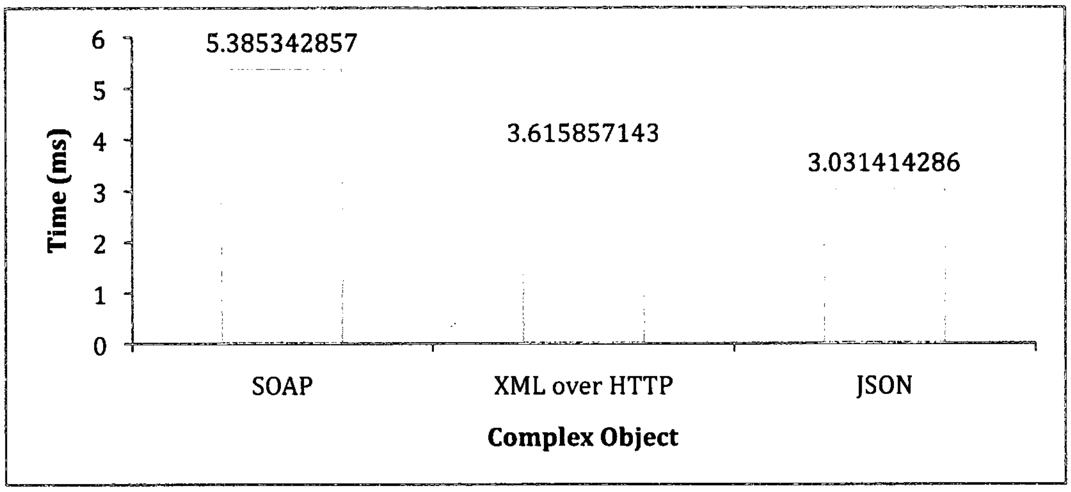


**Figure 29 – Experiment 2 Complex Object Average Round Trip Time**

Adding an array of 50 strings to the complex object better shows the processing time differences between the encoding mechanisms. While each shows an increase in processing time in Figure 30, those based on XML show a larger decrease in performance. XML over HTTP is also more clearly performing better than SOAP but not as well as JSON.

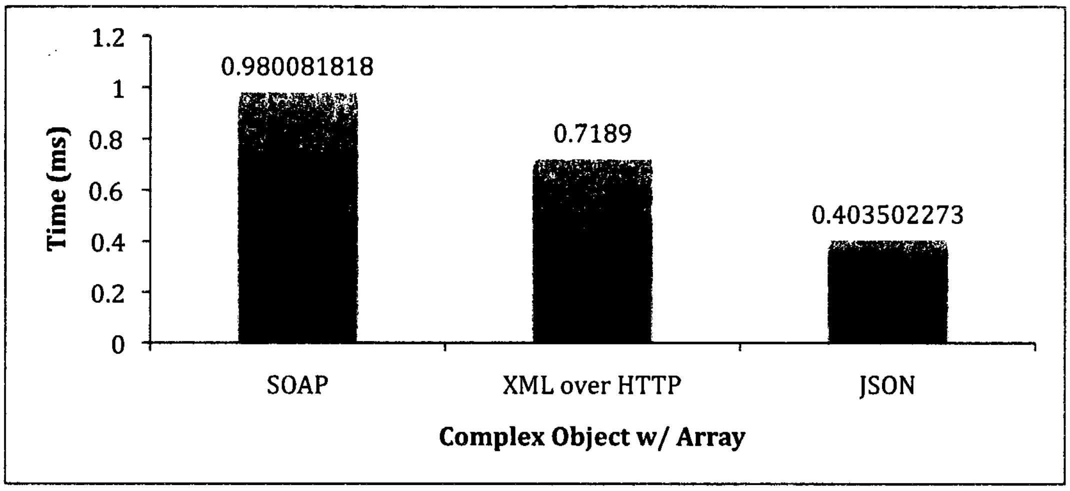


**Figure 30 – Experiment 2 Complex Object w/ Array Average Round Trip Time**

The remaining data type objects are made of the complex data type objects including an array, however they are composite objects of a defined depth. Figure 31 shows the performance characteristics of the data type objects with a depth of 5 layers. XML over HTTP becomes closer in performance to SOAP than JSON.

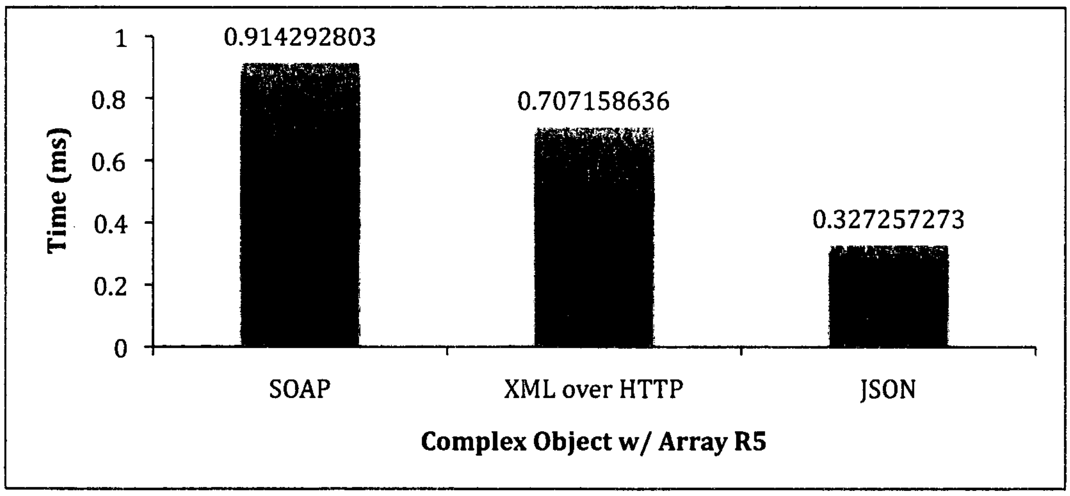


**Figure 31 - Experiment 2 Complex Object w/ Array R5 Average Round Trip Time**

The same trend can be seen in Figure 32 with a depth of 10 objects as with a depth of 5 objects.

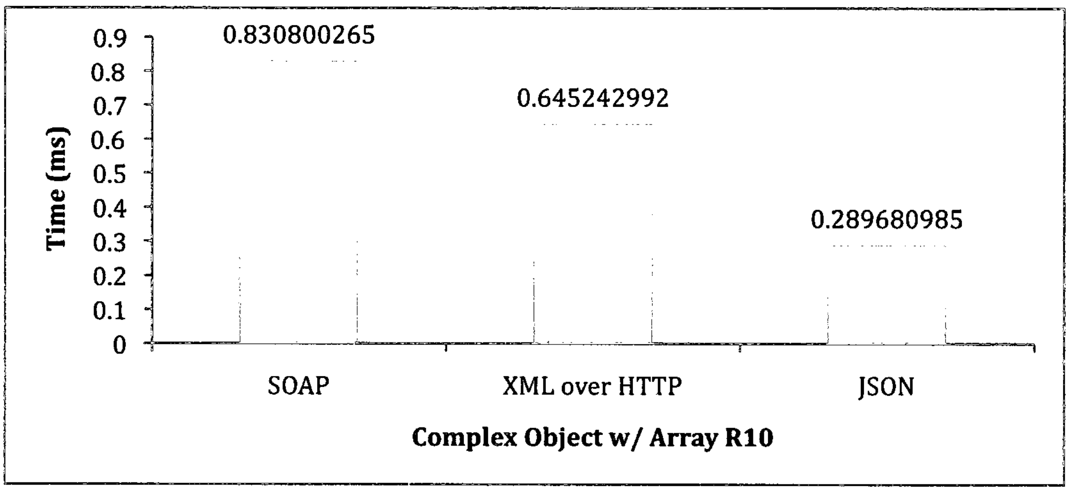


**Figure 32 - Experiment 2 Complex Object w/ Array R10 Average Round Trip Time**

A depth of 20 shows a performance roughly double that as a depth of 10 in Figure 33.

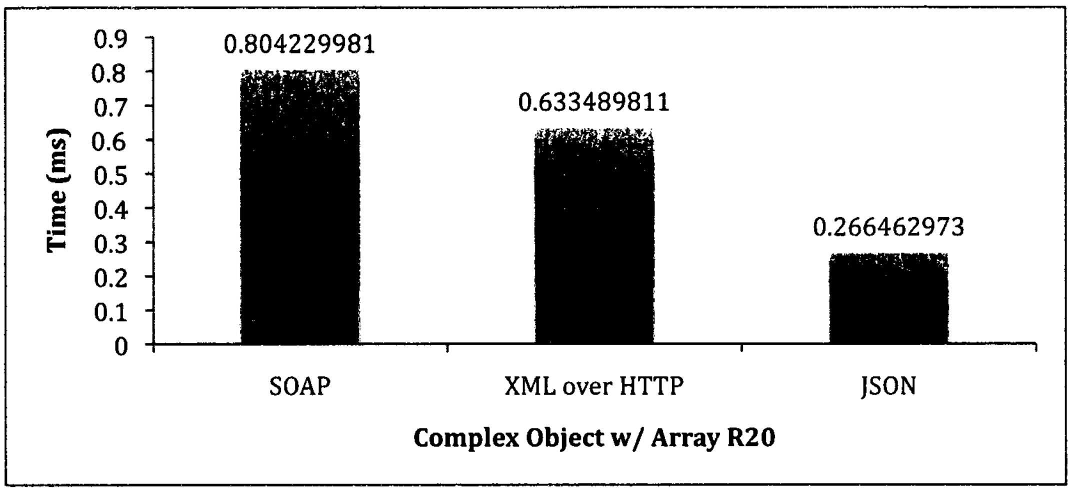


**Figure 33 - Experiment 2 Complex Object w/ Array R20 Average Round Trip Time**

The last object tested in Experiment 2 is of a data type object 40 layers in depth, performance characteristics shown in Figure 34. The same trend is carried through to this object. JSON still outperforms SOAP and XML over HTTP significantly.

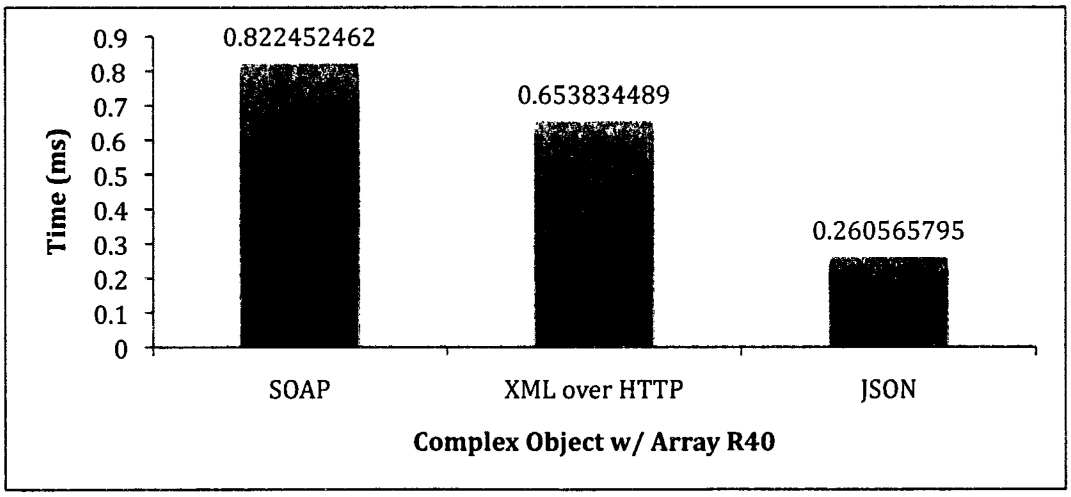


**Figure 34 - Experiment 2 Complex Object w/ Array R40 Average Round Trip Time**

### *5.2.2 Serialized Object Size Analysis*

The performance of each encoding mechanism must take into account more than CPU processing time. Network bandwidth usage must also be considered. To take this factor into account, the size of the serialized object is examined between the encoding mechanisms to identify which protocols utilize available bandwidth more efficiently.

Figure 35 shows the number of characters required to represent the data object using either SOAP or JSON for Experiment 1. Note the number of characters is substantially larger when using SOAP versus the JSON encoding mechanism.

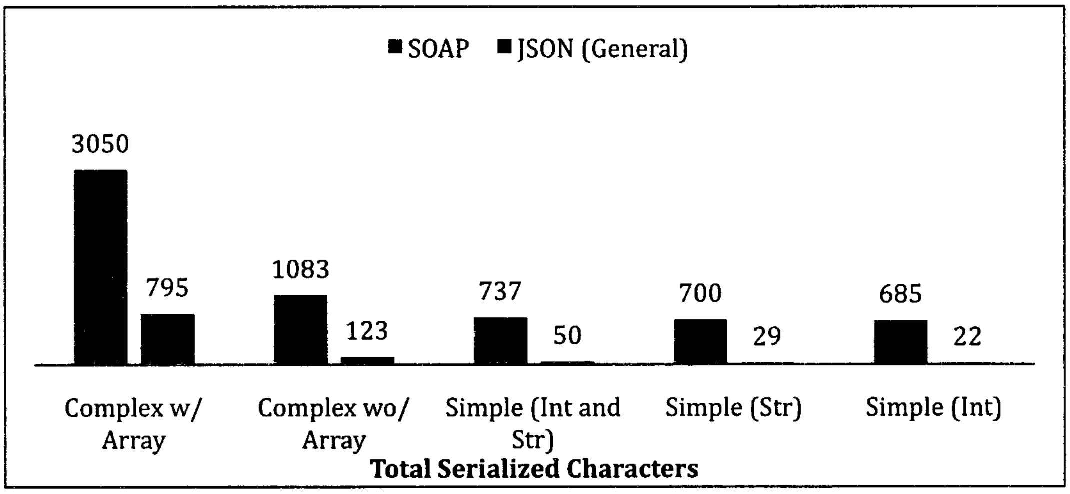


**Figure 35 – Experiment 1 Total Serialized Characters Comparison – SOAP and JSON**

The following figures show the character size of the protocols used in Experiment 2 for a data type containing a simple integer primitive. Note that SOAP requires more characters to represent the same data.

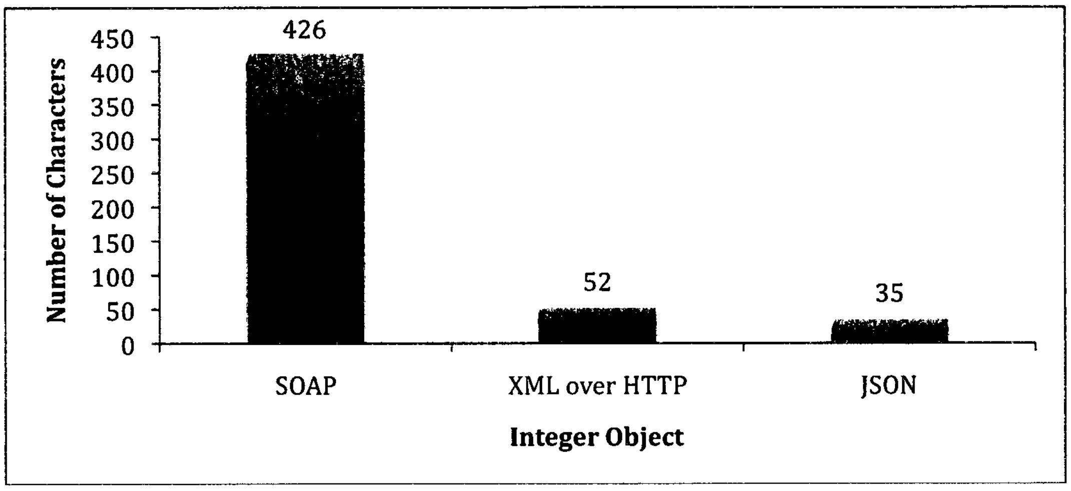


**Figure 36 – Experiment 2 Integer Object Response Result Characters Per Protocol**

The same trend can be seen in Figure 37 for a data type object containing a string value.

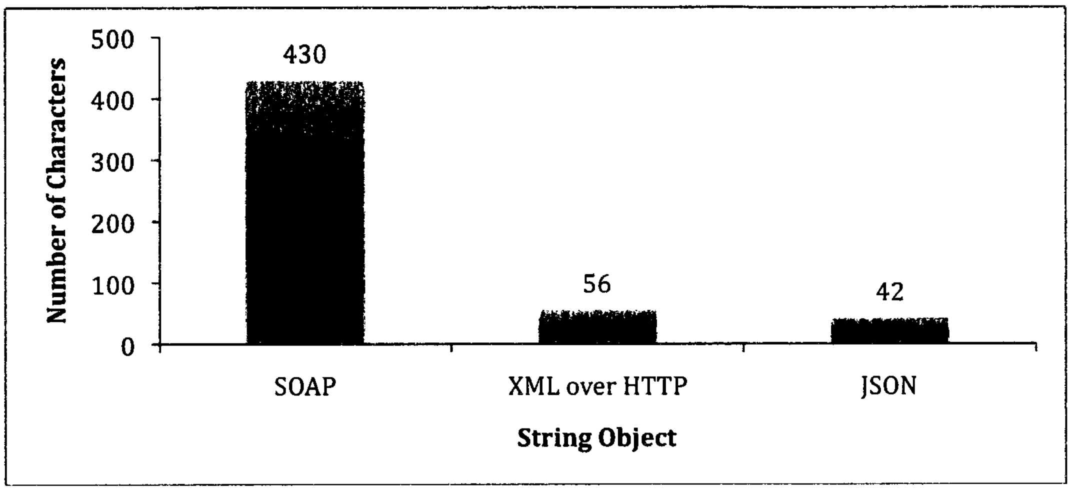


**Figure 37 - Experiment 2 String Object Response Result Characters Per Protocol**

Again, the trend stays the same with a data type object containing both a string and an integer value as shown in Figure 38.

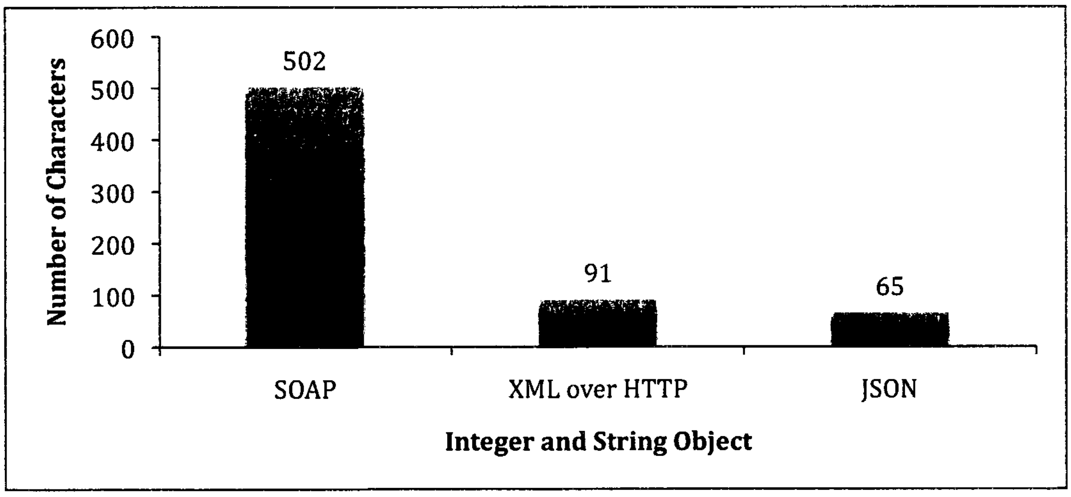


**Figure 38 - Experiment 2 Integer and String Object Response Result Characters Per Protocol**

A complex data type object requires more characters, and is the first data type in which JSON shows a significant character difference over XML over HTTP in Figure 39.

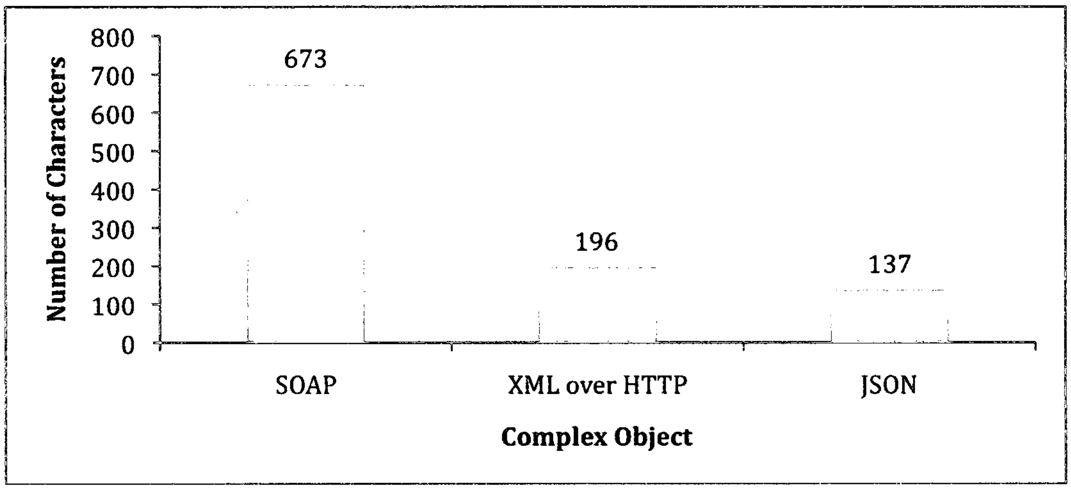


**Figure 39 - Experiment 2 Complex Object Response Result Characters Per Protocol**

Figure 40 shows what adding arrays to the data type objects can do to increase the character representation of the serialized objects. The trend is more pronounced. SOAP requires significantly more characters than XML over HTTP, with JSON requiring the fewest.

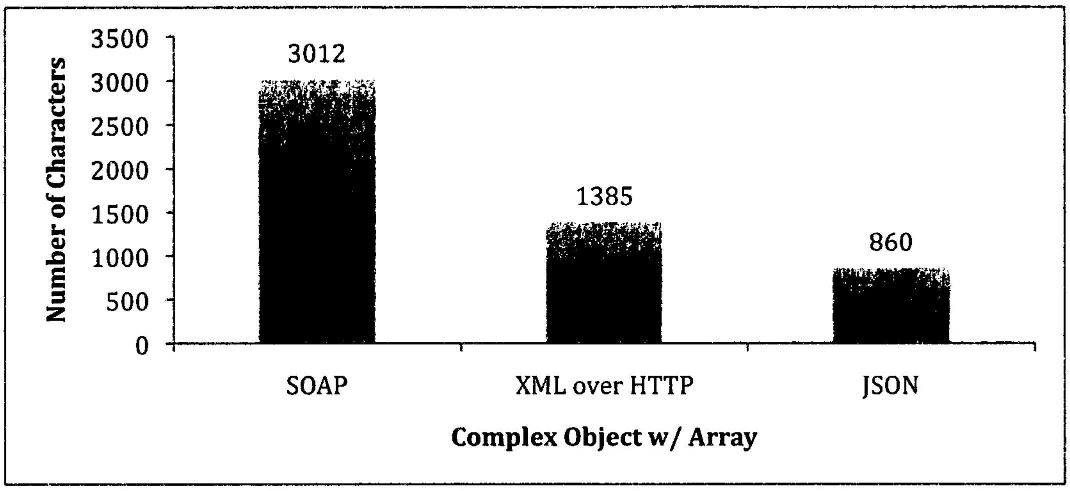


**Figure 40 - Experiment 2 Complex Object w/ Array Response Result Characters Per Protocol**

Objects of increasing depth indicate the same trend as less complex data types. Figure 41 shows the number of characters required to represent a data type object of 5 levels deep.

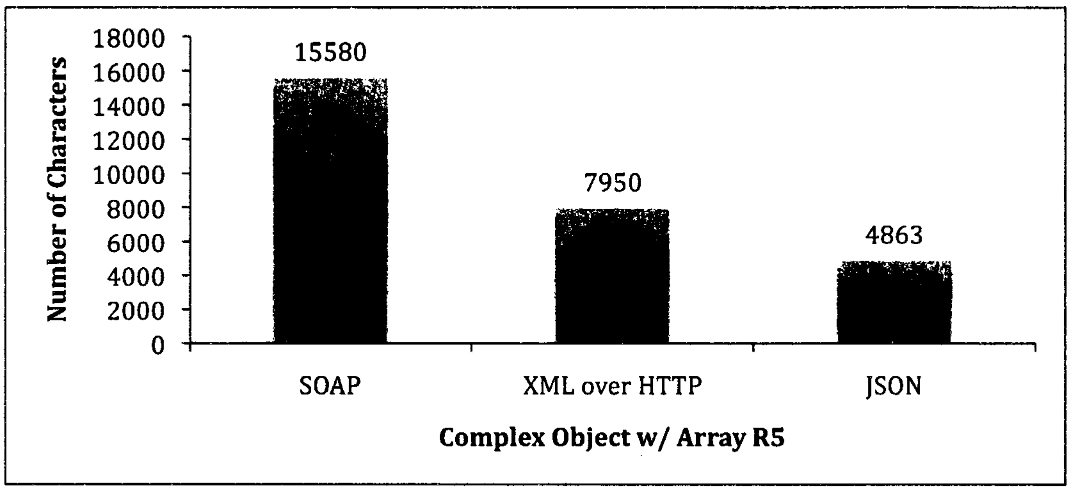


**Figure 41 - Experiment 2 Complex Object w/ Array Response Result Characters Per Protocol**

Figure 42 shows the number of characters required to represent an object 10 levels deep.

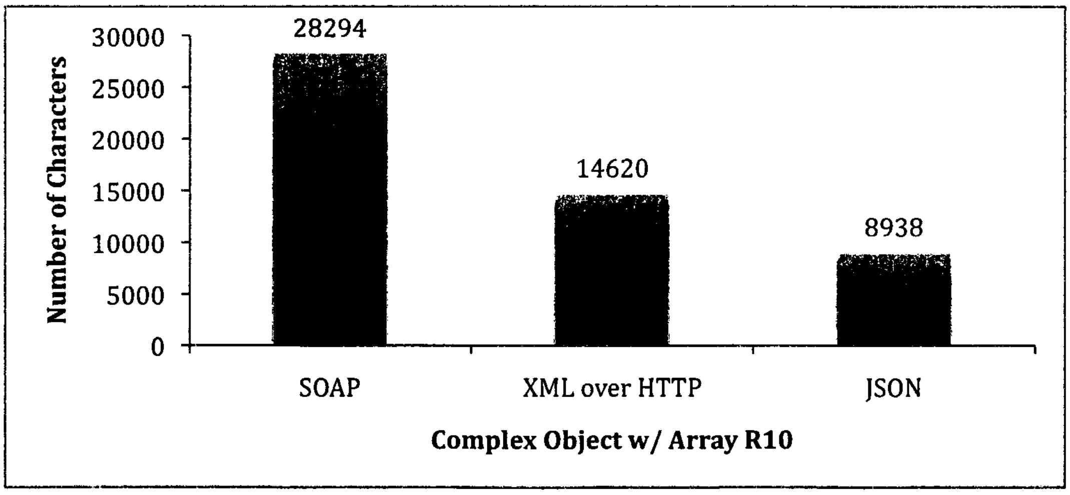


**Figure 42 - Experiment 2 Complex Object w/ Array R10 Response Result Characters Per Protocol**

A data type object of 20 levels deep is represented in Figure 43. The same identified trend continues as the object depth becomes greater.

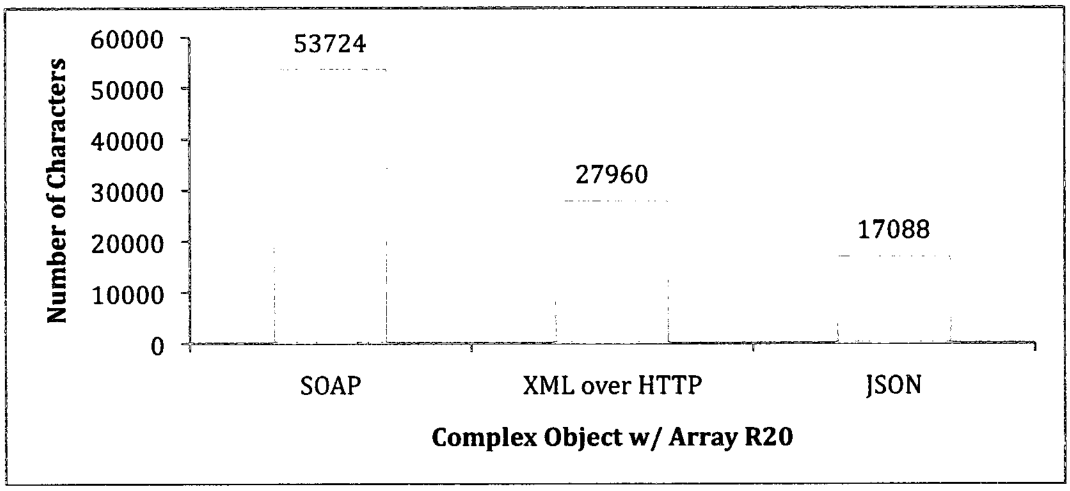


**Figure 43 - Experiment 2 Complex Object w/ Array R20 Response Result Characters Per Protocol**

The most complex object tested in Experiment 2, an object 40 layers deep, still shows the same characteristics as the complex object did. JSON is clearly the better performing encoding mechanism.

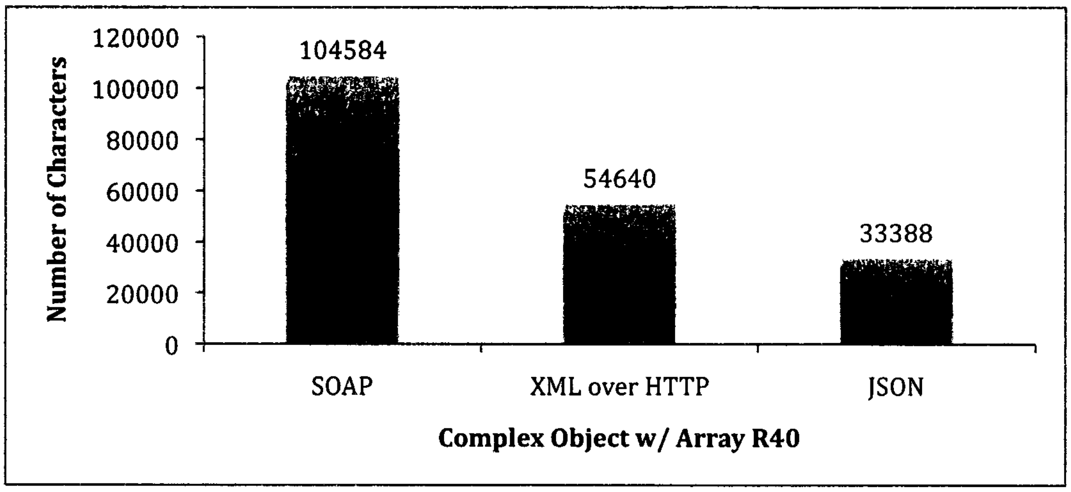


**Figure 44 - Experiment 2 Complex Object w/ Array R40 Response Result Characters Per Protocol**

### *5.2.3 Protocol Performance*

The sections above clearly show that JSON performs better in both processor time required to process the request as well as the number of characters required to represent the data type object in a format compatible with a web service. Comparing the time required to process the raw data object can identify the actual efficiency of the encoding method. Performing a time comparison of the output characters is not meaningful due to the nature of the encoding methods. For example, both SOAP and XML over HTTP utilize descriptive opening and closing tags, increasing the number of characters significantly when compared to JSON. Table 1 showed the amount of overhead required for each protocol. The substantial addition of characters in these encoding methods skews the numbers and is not an accurate method for measuring performance of the method.

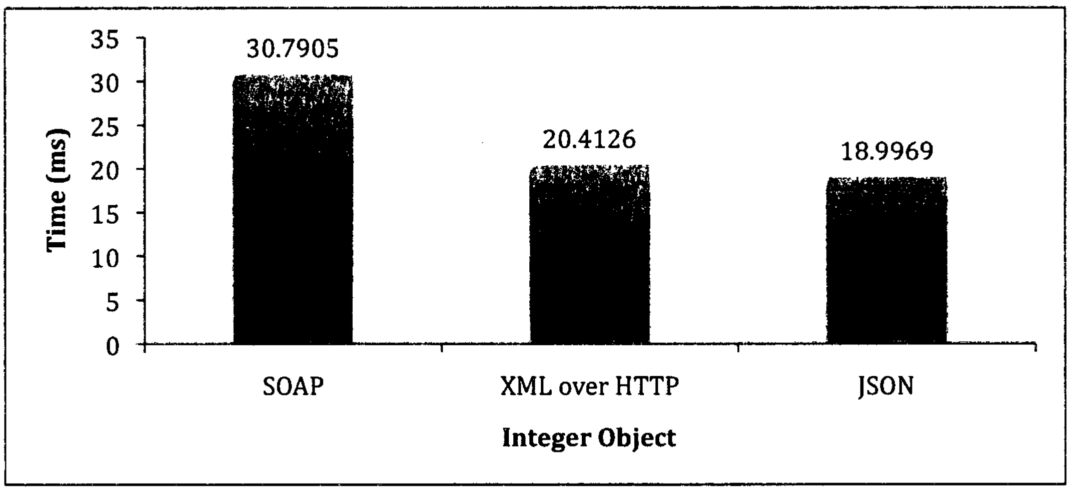


**Figure 45 – Integer Object Average Processing Time Per Raw Data Character**

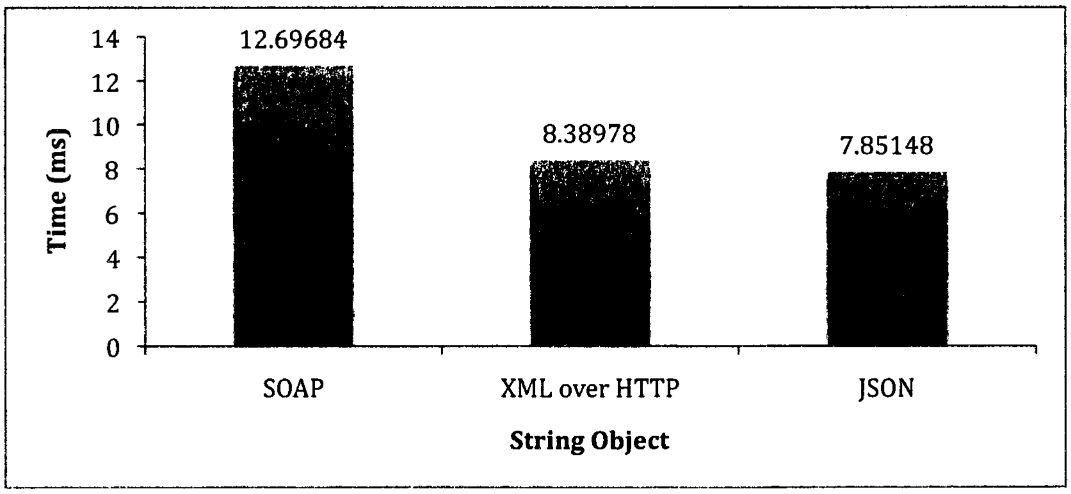


**Figure 46 – String Object Average Processing Time Per Raw Data Character**

Figures 45 and 45 above illustrate the performance of each protocol examined in Experiment 2 for the most simplistic data type objects examined. Note that JSON does perform better than the XML-based encoding methods that were examined, however the difference is negligible when compared to XML over HTTP. Of the encoding methods examined, SOAP performed the worst due to additional processing required for formatting. XML over HTTP does not require the additional header information that SOAP requires.

As shown in Figure 48, this trend continues, although the performance of JSON is shown to increase with more complicated data type objects. Adding additional data, an array of significant size as shown in Figure 49, shows that JSON performs even better than the XML-based protocols. All of the encoding methods appear to perform better due to the number of characters added to the data object by the array as shown by the average time per character processing.

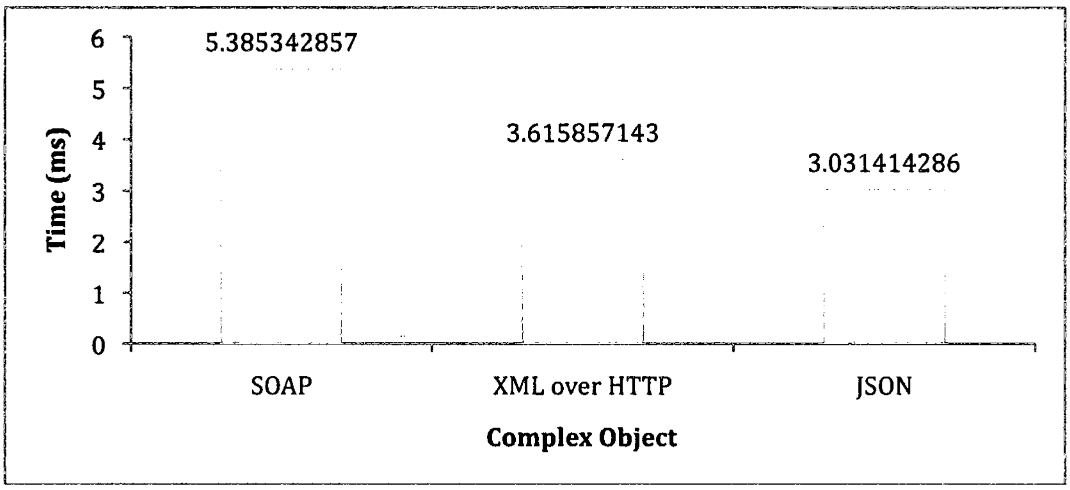


**Figure 47 – Complex Object Average Processing Time Per Raw Data Character**

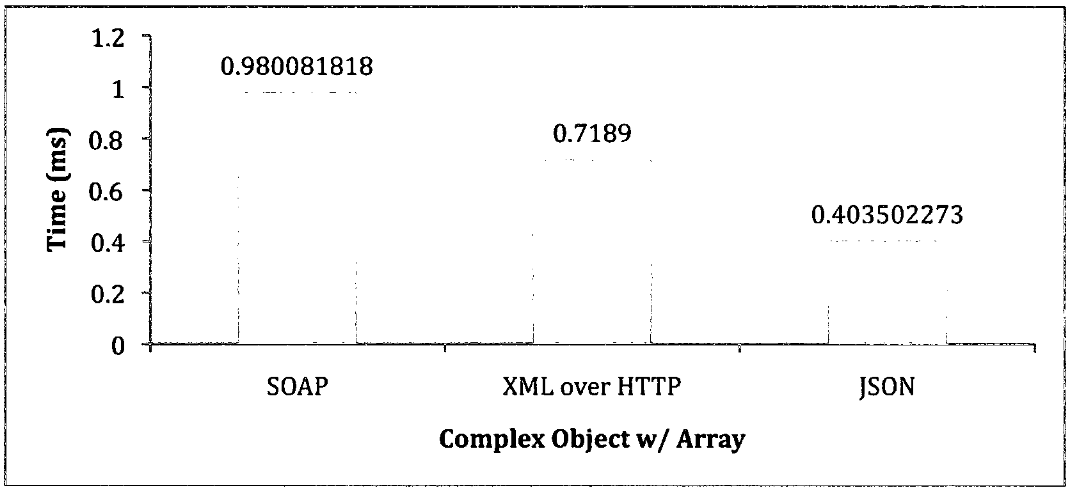


**Figure 48 – Complex Object w/ Array Average Processing Time Per Raw Data Character**

Figures 50 and 51 illustrate the performance per character of raw data for the largest data type objects tested in Experiment 2. The performance of JSON far exceeds that of the other encoding methods tested. Note that the data type object of 40 recursive objects performs better than that of 5 recursive objects. This is seen due to the number of characters of raw data being encoded.

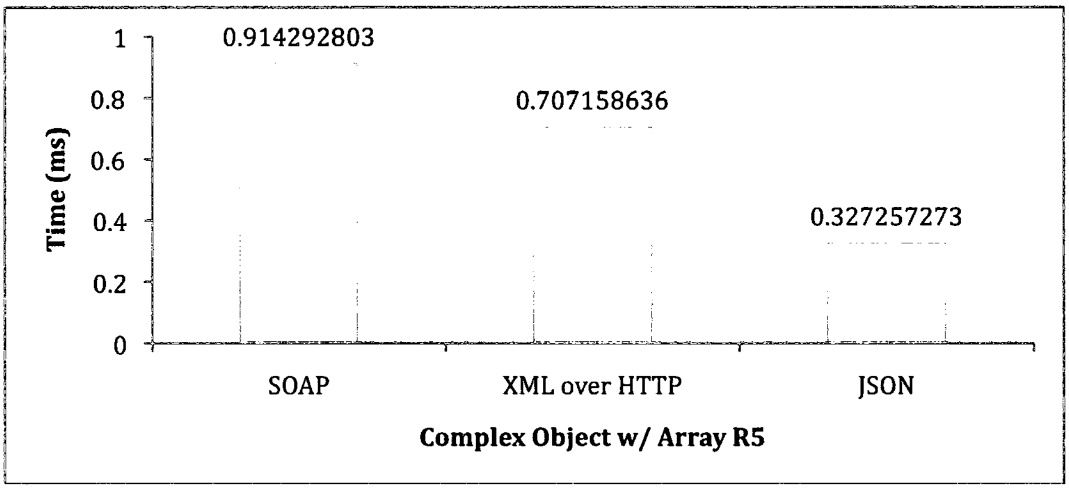


**Figure 49 – Complex Object w/ Array R5 Average Processing Time Per Raw Data Character**

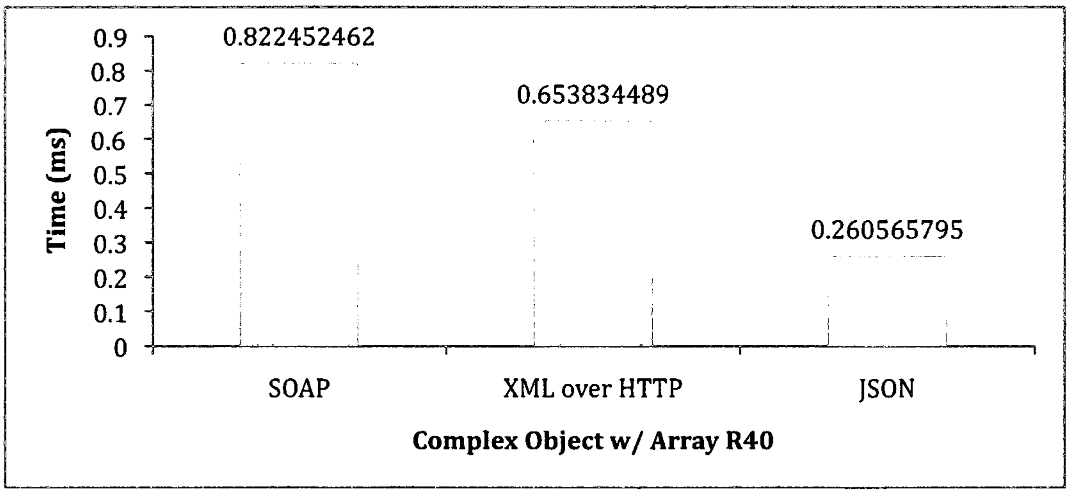


**Figure 50 – Complex Object w/ Array R40 Average Processing Time Per Raw Data Character**

## 5.3 Overall Analysis

Recall that two different resource utilization factors are being considered in the analysis. First, the processing time required for encoding and decoding the message being transferred through the web medium and secondly the used network bandwidth, measured by the number of response characters from the web service. Several web service protocols were used and measured throughout the experiments, some of these XML-based.

Examining Figure 8 and Figure 9, it is shown that in Experiment 1, either version of JSON demonstrated the use of fewer system resources needed for encoding or decoding an object. Figure 36 identifies the number of characters required to represent the response data in a particular algorithm. Notice that JSON required a significantly smaller number of characters to represent the same data.

Similar results were identified in Experiment 2. An additional protocol was introduced, XML over HTTP, to provide another XML-based protocol for comparison.

Results will now be broken down by measurement type. Following will be some analysis of the individual measurements and attempt to determine what factors affect the performance of each protocol.

## 5.4 Analysis By Processing Time

The encoding and decoding time can stress the resources of an embedded device significantly. The processor and memory allocations given to such a device are constrained to a degree that algorithms requiring a large number of CPU cycles or a significant amount of memory can cause the device to fail in its core function. Web services can use a number of transmission protocols, and finding one that is efficient on an embedded device can assist with reducing the system resource utilization overall for the device.

Two separate experiments were conducted, one to verify the idea that a significant performance difference can be seen on a full-featured computer system, and the second to implement a web services architecture on an embedded device and measure the performance of several protocols. This data would then be compared with the data collected in Experiment 1, acting as a baseline, to verify the performance on such a device.

Figures 8 and 9 identify the processing performance characteristics of serialization and deserialization of the five data set types being examined. Experiment 1 focused on the performance of a particular web service protocol in the Microsoft .Net environment. These two figures show that more complex objects require more processing time than those that are less complicated.

Figures 10 and 11 show the impact of serialization and deserialization of the data set types using SOAP, providing the average of the measured values for comparison. Again note how the more complicated the object, the more processing time is required to encode or decode. In addition, a trend can be seen that the amount of time required to serialize the data is much more significant than deserialization.

The JSON protocol, as shown in Figures 12 and 13, demonstrate the same trend. The more complicated the data structure, the longer it takes to process. When compared with the average SOAP timings per object, SOAP requires a significantly large amount of time. In converting the smallest data structure, one that contains an integer value as its only data type, JSON is capable of serializing the object 57 times faster than using SOAP. The most complication object, in this case a complex object with a 50-element array, the delta increases significantly with JSON serializing 132.4 times faster than SOAP.

Deserialization does not show the same trend. JSON is faster deserializing the smallest object by a magnitude of 7.5, however the largest object tested shows a much smaller performance difference at a magnitude of 1.19.

JSON Contract, the newer method that Microsoft implemented in their framework for handling JSON communication, performs better than the old implementation when examining the ability to serialize an object, as shown in Figures 14 and 15. The ability to deserialize an object using this method is slower with smaller objects. Larger objects show a decrease in deserialization time.

Figures 16 through 25 provide a view of serialization and deserialization performance per data set type. Note that each object demonstrates the same trends.

JSON performs better in both serialization and deserialization than SOAP, however the deserialization process does not show as much CPU utilization delta as the serialization process does.

Experiment 2 repeats the process followed in Experiment 1 with a few exceptions. Data is being transferred across a limited network infrastructure; therefore system and network latency is included in the calculated values. More importantly, the web service architecture is installed on an embedded device.

Each figure provides the average time required to transfer the request to the embedded device, deserialize the data, processes the received data, serialize the response, and send the response back to the client system. Network transfer time is accounted for by calculating the amount of time required to send a ping packet of equal or larger size, divided by two as the same data is sent on the return of the packet.

Figures 26 through 34 show the average time required to process a particular data type object using SOAP, XML over HTTP, and JSON. JSON performs better than both SOAP and XML over HTTP in all cases, however as the objects become larger, the performance delta between JSON and the other encoding schemes increase. Figure 26 shows the average time difference between JSON and XML over HTTP being low at a magnitude of 1.07, while the largest object not containing recursive objects shows a magnitude of 1.78. Once a data object using recursive objects is introduced, this changes to a magnitude of 2.16. The most deeply recursive data object shows a difference between JSON and XML over HTTP of a magnitude of 2.51.

## 5.5 Analysis By Character Size

The most critical aspect of a web service that affects a network is the amount of data that must be transferred from client to server and back. Compounded by the number of requests, a network can quickly become saturated with traffic slowing down all transfers. Reducing the amount of traffic across the network is important in preventing network failures due to congestion.

The character sizes for the serialized data type objects are shown in Figure 36 for both SOAP and JSON. No difference in character size between both implementations of JSON in the Microsoft .Net Framework was observed.

Figure 35 shows JSON being capable of encoding data in a much smaller package than SOAP is capable of. For example, a data object containing an integer primitive type containing a four-digit value encoded with JSON is represented using 22 characters, while a SOAP representation of an object instantiated from the same class requires 685 characters. In this case, the character reduction is a magnitude of 31.14, showing a significant improvement. The largest data type used in Experiment 1 shows an improvement of 3.84 in magnitude. The larger the object is, more characters are required for representation.

Data collected in Experiment 2 is shown in Figures 36 through 44. An expanded set of data objects was used, as well as an additional protocol. The delta of characters required to represent an object is decreased as the object becomes larger in size. XML over HTTP resides between SOAP and JSON in all cases.

A difference is seen in the size of objects between Experiment 1 and 2, caused by the implementations of the SOAP and JSON protocols used in each case. The web services architecture used on the embedded device wraps responses within a response tag as part of its error handling mechanism, to ensure the client is able to process the return message. While the response received may not be as expected, the client can determine that it did receive a response from the web service and process the received error message accordingly. The difference between the SOAP and XML over HTTP character size can be attributed to the header information within the SOAP message. This header can be formatted with varying options, which can increase or decrease its size. The embedded device uses a more simplistic SOAP header, accounting for much of the character size difference between the two experiments.

## 5.6 Proof Criteria Completion

To determine that the research has been completed, several proof criteria have been defined. For convenience, these criteria are listed below:

- The selected encoding methods from both experiments show a trend indicating that methods not using XML require less processing time for each data object type.
- The selected encoding methods from both experiments show that methods not using XML require less network bandwidth for each data object type.
- The selected encoding methods implemented on an embedded device, which do not use XML, indicate higher performance per character input.

Table 4 represents the number of milliseconds required to encode and decode each of the tested data object types in Experiment 1 using SOAP. When compared with the values in Tables 5 and 6 representing the time required to encode and decode each data object type in Experiment 1 using differing versions of JSON, it is clear that JSON encodes and decodes using less time than SOAP.

**Table 4 – Experiment 1 SOAP Summary**

| Object Type | Average Serialization (Milliseconds) | Average Deserialization (Milliseconds) | Output Characters |
|---|---|---|---|
| Complex w/ Array | 12.00013502 | 0.837155337 | 3050 |
| Complex w/o Array | 1.704639839 | 0.271917447 | 1083 |
| Simple (Int and Str) | 0.938253166 | 0.18467602 | 737 |
| Simple (Str) | 0.908111497 | 0.172314372 | 700 |
| Simple (Int) | 0.806270995 | 0.165810935 | 685 |

**Table 5 – Experiment 1 JSON Summary**

| Object Type | Average Serialization (Milliseconds) | Average Deserialization (Milliseconds) | Output Characters |
|---|---|---|---|
| Complex w/ Array | 0.090605355 | 0.704883773 | 795 |
| Complex w/o Array | 0.03300914 | 0.100295994 | 123 |
| Simple (Int and Str) | 0.01562418 | 0.042960479 | 50 |
| Simple (Str) | 0.010695108 | 0.024706664 | 29 |
| Simple (Int) | 0.014148983 | 0.022093739 | 22 |

**Table 6 – Experiment 1 JSON Contract Summary**

| Object Type | Average Serialization (Milliseconds) | Average Deserialization (Milliseconds) | Characters To Send |
|---|---|---|---|
| Complex w/ Array | 0.068785003 | 0.179703326 | 795 |
| Complex w/o Array | 0.016200927 | 0.063464241 | 123 |
| Simple (Int and Str) | 0.012336989 | 0.044223644 | 50 |
| Simple (Str) | 0.009486394 | 0.040443661 | 29 |
| Simple (Int) | 0.010529954 | 0.041021911 | 22 |

Similar results can be seen in Tables 7 through 15 which summarize the data from Experiment 2, providing mean averages of the time to process a specific data type object as well as its mean average per source character encoded. The processing time for each object starts after the client application submits the request for the data type object to the embedded device and ends once the object has been received.

**Table 7 – Experiment 2 Integer Object Summary**

| | SOAP | XML over HTTP | JSON |
|---|---|---|---|
| Result Character Size | 426 | 52 | 35 |
| Data Character Size | 4 | 4 | 4 |
| Avg Time Per Object (ms) | 123.162 | 81.6504 | 75.9876 |
| Avg Time Per Data Char (ms) | 30.7905 | 20.4126 | 18.9969 |

Table 8 – Experiment 2 String Object Summary

| | SOAP | XML over HTTP | JSON |
|---|---|---|---|
| Result Character Size | 430 | 56 | 42 |
| Data Character Size | 10 | 10 | 10 |
| Avg Time Per Object (ms) | 126.9684 | 83.8978 | 78.5148 |
| Avg Time Per Data Char (ms) | 12.69684 | 8.38978 | 7.85148 |

Table 9 – Experiment 2 Integer and String Object Summary

| | SOAP | XML over HTTP | JSON |
|---|---|---|---|
| Result Character Size | 502 | 91 | 65 |
| Data Character Size | 14 | 14 | 14 |
| Avg Time Per Object (ms) | 132.4284 | 88.0932 | 76.674 |
| Avg Time Per Data Char (ms) | 9.459171429 | 6.292371429 | 5.476714286 |

Table 10 – Experiment 2 Complex Object Summary

| | SOAP | XML over HTTP | JSON |
|---|---|---|---|
| Result Character Size | 673 | 196 | 137 |
| Data Character Size | 28 | 28 | 28 |
| Avg Time Per Object (ms) | 150.7896 | 101.244 | 84.8796 |
| Avg Time Per Data Char (ms) | 5.385342857 | 3.615857143 | 3.031414286 |

Table 11 - Experiment 2 Complex Object With Array Summary

| | SOAP | XML over HTTP | JSON |
|---|---|---|---|
| Result Character Size | 3012 | 1385 | 860 |
| Data Character Size | 528 | 528 | 528 |
| Avg Time Per Object (ms) | 517.4832 | 379.5792 | 213.0492 |
| Avg Time Per Data Char (ms) | 0.980081818 | 0.7189 | 0.403502273 |

Table 12 – Experiment 2 Complex Object With Array R5 Summary

| | SOAP | XML over HTTP | JSON |
|---|---|---|---|
| Result Character Size | 15580 | 7950 | 4863 |
| Data Character Size | 2640 | 2640 | 2640 |
| Avg Time Per Object (ms) | 2413.733 | 1866.8988 | 863.9592 |
| Avg Time Per Data Char (ms) | 0.914292803 | 0.707158636 | 0.327257273 |

Table 13 - Experiment 2 Complex Object With Array R10 Summary

| | SOAP | XML over HTTP | JSON |
|---|---|---|---|
| Result Character Size | 28294 | 14620 | 8938 |
| Data Character Size | 5280 | 5280 | 5280 |
| Avg Time Per Object (ms) | 4386.6254 | 3406.883 | 1529.5156 |
| Avg Time Per Data Char (ms) | 0.830800265 | 0.645242992 | 0.289680985 |

Table 14 - Experiment 2 Complex Object With Array R20 Summary

| | SOAP | XML over HTTP | JSON |
|---|---|---|---|
| Result Character Size | 53724 | 27960 | 17088 |
| Data Character Size | 10560 | 10560 | 10560 |
| Avg Time Per Object (ms) | 8492.6686 | 6689.6524 | 2813.849 |
| Avg Time Per Data Char (ms) | 0.804229981 | 0.633489811 | 0.266462973 |

Table 15 - Experiment 2 Complex Object With Array R40 Summary

| | SOAP | XML over HTTP | JSON |
|---|---|---|---|
| Result Character Size | 104584 | 54640 | 33388 |
| Data Character Size | 21120 | 21120 | 21120 |
| Avg Time Per Object (ms) | 17370.196 | 13808.9844 | 5503.1496 |
| Avg Time Per Data Char (ms) | 0.822452462 | 0.653834489 | 0.260565795 |

As shown by these tables, JSON outperforms XML over HTTP and SOAP in every case, both in the resulting character size and mean average to process each data type object. The mean time per data character within the data type object indicates that JSON is higher performing as well.

The summarized data indicates that all proof criteria have been satisfied. JSON outperforms SOAP in every case in both network bandwidth required to represent the object and in processing time, as well as being an encoding method that does not formatted using XML.

## 5.7 Rejection Of Null Hypothesis

The null hypothesis for this research was stated above as:

**$H_0$:** Web services using an XML-based protocol would see the same or higher system resource utilization by using a transmission protocol that is not XML-based.

The results of the experimental research demonstrate that the null hypothesis is rejected for all combinations of data types and web service encodings that were tested.

## 5.8 Acceptance Of Alternative Hypothesis

Based on the rejection of the null hypothesis for all combinations of data types and encodings evaluated in this research, the following alternative hypothesis is accepted:

**Ha:** Web services can reduce system resource utilization by using a transmission protocol that is not XML-based.

## 5.9 Other Observations

The following observations have been identified during the course of this research.

### *5.9.1 Embedded Web Service*

Web services on embedded devices has been implemented using gSOAP [12], however it is not a flexible implementation. All code is written in C++ which requires code compilation outside the device, or if the device hosts an operating system, on the device.

With the web services architecture developed within Experiment 2, a web service architecture can be implemented providing dynamic customization of web services with little downtime on the device. The user, not requiring a set of development tools and significant downtime for development, can use such a feature to create a web-enabled device that is completely customizable.

### *5.9.2 XML In Web Services*

Both Experiments 1 and 2 show better performing protocols exist that can be supported on both a full computer system as well as an embedded device. The key difference between the protocols is the use of XML as the base. JSON does not use XML, which can be wordy with the formal but self-describing opening and closing tags. While beneficial for reading the document, it does decrease performance.

This can be verified in Experiment 2 as JSON performed better than both XML over HTTP and SOAP. XML over HTTP does not use as large of a header as SOAP, but the verbosity of the tags and the mechanisms for processing an XML document shows an evident performance impact.

## 5.10 Summary

Service Oriented Architecture technologies are powerful tools, allowing organizations to develop scalable and flexible solutions for their environments. In the industrial controls world, however, such technology breakthroughs are non-existent. In these cases, proprietary software and application programming interface (API) are required to manipulate these devices, which can be different from device to device.

Concern exists regarding the performance of the encoding mechanisms that are used in web service environments, namely because of the use of XML. While XML allows for the creation of a human-readable message, the overhead involved with the formatting increases the network traffic as well as the processing time due to the methods that exist to validate XML.

This research proves there are more efficient encoding methods that could be used with Service Oriented Architectures as a mainstream transmission method. It also shows that flexible web service frameworks can be used on an embedded device using a variety of encoding mechanisms providing a universal communication method with these devices.

The use of the JSON encoding mechanism in this research demonstrated it was higher performing than XML-based encoding mechanisms in use as transmission methods for web services. This observation was seen in all test cases examined throughout the course of this research. The delta between performance of each encoding method varied significantly from over one and a half times to several hundred times in specific cases. More importantly, it has been shown a web service architecture can be implemented on a resource-constrained device.

# CHAPTER VI

# CONCLUSIONS

This chapter provides a high-level summary of the results as well as a discussion identifying an example where the resulting research can be used in a system as well act as an enabler for intelligent systems.

## 6.1 Results

Service Oriented Architectures (SOA) are powerful constructs that can lead to complex distributed systems using common communication mechanisms, which is lacking in industrial embedded applications. The ability for industrial devices to be able to talk to each other using a common transmission can reduce incompatibility between products, the amount of time to implement solutions using embedded industrial devices, and simplify troubleshooting of communication issues. This research provides the basis to develop solutions that can use a consistent communication method, which is already used in other industries.

Several encoding mechanisms that are traditionally used with embedded devices were analyzed in this research with the goal of discovering which of these methods performs the best on an embedded device. Embedded devices have fewer resources available to them when compared to a full computing system, therefore the characteristics of the encoding method can prove to be significantly different when executed in the new environment. This research validates this is not the case with the mechanisms that were tested (SOAP, XML over HTTP, and JSON) as the performance of each method scales to the embedded device used for experimentation. In all tested cases, JSON outperformed SOAP and XML over HTTP in both the network bandwidth required to transmit data as well as the amount of time to encode, transmit, and decode the data.

## 6.2 Significance

This research illustrates that building an industrial control, which uses a web service communication mechanism, is feasible for environments where a consistent communication method is advantageous. Web service architectures not only allow the devices to communicate with each other, but will allow any computing device to communicate with them as well. Industrial controls will finally be able to communicate using the same methods that distributed applications use in the Enterprise. When partnered with more capable devices, the devices can make intelligent decisions based on input from other intelligent devices on the network. Currently, this has not been done in the industrial control industry and will significantly increase the capabilities of the devices used in industrial environments.

Imagine an industrial environment using products from varying vendors creating a unified solution in which every device is able to talk to each other using the same communication technologies. A device that monitors the level of fluid in a tank can communicate with a device that controls a pump that feeds the tank. Another device, which uses the fluid in the tank, can talk to the tank controller as well as other industrial devices to predict how many more products it can create before running out of resources. The resulting data can be used to trigger an alert letting a human operator know when they will need to provide additional resources to the system otherwise risk loss of productivity. Each of the devices in this example may come from a different vendor (one specializing in tank controls, another specializing in assembly line devices), however they are able to communicate together.

Taking the example another step, giving each of the devices some intelligence can create a self-maintaining system. Giving each device more memory and processing horsepower can support a solution in which each device has its own artificial intelligence and is capable of determining what is best in an array of situations the device may experience. Such solutions will vastly surpass the capabilities of ladder logic currently used with PLC devices.

These examples can become reality, using existing open-source tools, which are readily available. This research provides a framework that can be used to build web-enabled industrial control solutions on resource-constrained devices opening the industry to standards-based communication mechanisms that provide interoperability with other web-enabled devices, including those not specific to the industrial control industry. Consistent and open communication mechanisms remove proprietary communication constraints, which currently exist, and allow industrial control vendors to focus on the features of the device. This kind of freedom will lead to devices that are significantly more advanced than those which are currently "state of the art."

# CHAPTER VII

# AREAS FOR FUTURE RESEARCH

## 7.1 Devices With Fewer Resources

This research utilized a device that provided more resources than most embedded devices used for industrial solutions. Sensor network motes, for example, have vastly less memory, processing capabilities, and storage due to their power requirements. The implementation of web service architectures on such devices using an efficient protocol such as JSON could yield a non-proprietary communications method enabling easier expandability and cross vendor sensor network implementations. Further investigation must be done to verify web service functionality on these devices can be achieved with adequate response. Investigation of encoding mechanisms can be performed on such devices as well, identifying performance characteristics on a class of devices with even fewer resources, as the research described here provides a proof of concept for such implementations.

## 7.2 Other protocols

The encoding methods used in this research represent only a small subset of available encodings available; although methods exist that have not been utilized in a web service environment. Additional human readable encoding mechanisms could show promise as general web service protocols and allow for higher performing infrastructures.

## 7.3 Intelligent Control Devices

Throughout this research, a portion of a framework has been developed that could be used to expand the capabilities of control devices. Using an embedded device and web service architecture together, it is possible to create an intelligent device capable of various industrial/commercial or home automation tasks, which can be controlled via a web browser without requiring proprietary frameworks for communication.

As an example, a home or business climate control device could be designed allowing the operator to view and adjust climate settings within a building using any web browser. Such a device can acquire weather reports and humidity indexes for the area and include the captured information in its process for achieving the desired conditions the user has set.

Another example, more for industrial manufacturing environments, would be a device or series of devices capable of monitoring inventory of components and would manufacture the needed inventory for open orders that can be processed during the day, eliminating a substantial amount of inventory that would otherwise be required.

## 7.4 Object-Based Control System

The framework discussed and developed throughout this research can be used as a basis for an object-oriented control system using embedded devices. Such a control system can pass objects to a central monitoring station, or in the case of a sensor network between each mote on the network. Such a solution can make it easier for developers to interact with the system, as they are able to use object-oriented concepts and apply them to device communication. Objects defined in the web service can be used in their application code without the additional development effort required to parse data through the network stack.

Most popular programming languages provide facilities to communicate with a web service and more developers are becoming familiar with this paradigm. An infrastructure utilizing web services is much easier for a developer to become accustomed to when compared to learning a custom, often-proprietary communication method. Such a solution would allow a programmer to communicate to an embedded device in the same way they would any other web service provider.

## REFERENCES

1. Erl, T., *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. 2007: Prentice Hall PTR.

2. *Home*. 2009 [cited 2009 July 8]; Available from: http://labjack.com/.

3. *Data Acquisition Product Selection Guide*. 2009 [cited 2009 July 8]; Available from: http://www.mccdaq.com/products/product-selection.aspx.

4. Netburner. *Embedded Control*. 2009 [cited 2009 May 22]; Available from: http://www.netburner.com/embedded_control.html.

5. Technologic. *TS-7800 Embedded Computer - Rugged & Reliable, 500MHz, FPGA*. 2008 [cited 2008 December 2]; Available from: http://www.embeddedarm.com/products/board-detail.php?product=TS-7800.

6. Bonfatti, F., G. Gadda, and P.D. Monari, *Re-usable software design for programmable logic controllers*, in *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, \& tools for real-time systems*. 1995, ACM Press: La Jolla, California, United States. p. 31-40.

7. Prinsloo, J.M., et al., *A Service Oriented Architecture for Wireless Sensor and Actor Network Applications*. Mobile Networks and Applications, 2006. **11**(4): p. 469-485.

8. Bonivento, A., L.P. Carloni, and A. Sangiovanni-Vincentelli, *Platform-based design of wireless sensor networks for industrial applications*, in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*. 2006, European Design and Automation Association: Munich, Germany. p. 1103-1107.

9. Bortolazzi, J., T. Hirth, and T. Raith, *Specification and design of electronic control units*, in *Proceedings of the conference on European design automation*. 1996, IEEE Computer Society Press: Geneva, Switzerland. p. 36-41.

10. Engelen, R.v., *Code generation techniques for developing light-weight XML Web services for embedded devices*, in *Proceedings of the 2004 ACM symposium on Applied computing*. 2004, ACM Press: Nicosia, Cyprus. p. 854-861.

11. *gSOAP: SOAP C++ Web Services*. Available from: http://www.cs.fsu.edu/~engelen/soap.html.

12. Zeeb, E., et al., *Service-Oriented Architectures for Embedded Systems Using Devices Profile for Web Services*, in *21st International Conference on Advanced Information Networking and Applications Workshops*. 2007. p. 7.

13. Fummi, F., et al., *Interactive presentation: A middleware-centric design flow for networked embedded systems*, in *Proceedings of the conference on Design, automation and test in Europe*. 2007, EDA Consortium: Nice, France. p. 1048-1053.

14. Greenstein, B., E. Kohler, and D. Estrin, *A sensor network application construction kit (SNACK)*, in *Proceedings of the 2nd international conference on Embedded networked sensor systems*. 2004, ACM Press: Baltimore, MD, USA. p. 69-80.

15. Hahn, J., Q. Xie, and P.H. Chou, *Rappit: framework for synthesis of host-assisted scripting engines for adaptive embedded systems*, in *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. 2005, ACM Press: Jersey City, NJ, USA. p. 315-320.

16. Hakala, I. and T. Merj, *From vertical to horizontal architecture: a cross-layer implementation in a sensor network node*, in *Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*. 2006, ACM Press: Nice, France.

17. Hill, J., et al., *System architecture directions for networked sensors*, in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*. 2000, ACM Press: Cambridge, Massachusetts, United States. p. 93-104.

18. Intanagonwiwat, C., R. Govindan, and D. Estrin, *Directed diffusion: a scalable and robust communication paradigm for sensor networks*, in *Proceedings of the 6th annual international conference on Mobile computing and networking*. 2000, ACM Press: Boston, Massachusetts, United States. p. 56-67.

19. Matelan, M.N., *Automating the design of microprocessor-based real time control systems*, in *Proceedings of the 13th conference on Design automation*. 1976, ACM Press: San Francisco, California, United States. p. 462-469.

20. Stavroulaki, V., et al., *Distributed web-based management framework for ambient reconfigurable services in the intelligent environment*. Mobile Networks and Applications, 2006. **11**(6): p. 889-900.

21. Yalagandula, P., et al., *S3: a scalable sensing service for monitoring large networked systems*, in *Proceedings of the 2006 SIGCOMM workshop on Internet network management*. 2006, ACM Press: Pisa, Italy. p. 71-76.

22. Levis, P. and D. Culler, *Maté: a tiny virtual machine for sensor networks*, in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. 2002, ACM Press: San Jose, California. p. 85-95.

23. Hashmi, N., et al., *A sensor-based, web service-enabled, emergency medical response system*, in *Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services*. 2005, USENIX Association: Seattle, Washington. p. 25-29.

24. He, T., et al., *VigilNet: An integrated sensor network system for energy-efficient surveillance*. ACM Trans. Sen. Netw., 2006. **2**(1): p. 1-38.

25. Lennon, W.J., et al., *Using a distributed mini-computer network to automate a biochemical laboratory*, in *Proceedings of the ACM SIGMINI/SIGPLAN interface meeting on Programming systems in the small processor environment*. 1976, ACM Press: New Orleans, Louisiana, United States. p. 156-162.

26. Fielding, R.T. and R.N. Taylor, *Principled Design of the Modern Web Architecture*. ACM Trans. Inter. Tech., 2002. **2**(2): p. 35.

27. Erenkrantz, J.R., et al. *From Representations to Computations: The Evolution of Web Architectures*. in *Proceedings of the 6th joing meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 2007. Dubrovnik, Coratia.

28. Gannod, G.C., J.E. Burge, and S.D. Urban, *Issues in the Design of Flexible and Dynamic Service-Oriented Systems*, in *International Conference on Software Engineering*. 2007.

29. Jazayeri, M., *Some Trends in Web Application Development*, in *International Conference on Software Engineering*. 2007, IEEE Computer Society. p. 199-213.

30. Huynh, D.F., D.R. Karger, and R.C. Miller, *Exhibit: Lightweight Structured Data Publishing*, in *International World Wide Web Conference*. 2007, ACM. p. 737-746.

31. Jammes, F., A. Mensch, and H. Smit, *Service-oriented Device Communication Using the Devices Profile for Web Services*, in *ACM International Conference Proceeding Series*. 2005: Grenoble, France. p. 1-8.

32. Anh Phan, K., Z. Tari, and P. Bertok, *A Benchmark on SOAP's Transport Protocols Performance for Mobile Applications*, in *Symposium on Applied Computing*. 2006, ACM: Dijon, France. p. 1139-1144.

33. Head, M.R., et al., *Benchmarking XML Processors for Applications in Grid Web Services*, in *Conference on High Performance Networking and Computing*. 2006, ACM: Tampa, Florida.

34. Schmelzer, R., *Pros and Cons of XML*. 2001.

35. Lawrence, R., *The space efficiency of XML*. Information and Software Technology, 2004. **46**(11): p. 753-759.

36. Crockford, D., *The application/json Media Type for JavaScript Object Notation (JSON)*. 2006.

37. Ng, A. *Optimising Web Services Performance With Table Driven XML*. in *Software Engineering Conference*. 2006: IEEE Computer Society.

38. Ali, A., et al. (2002) *Web Services: Promises and Compromises*.

39. *SOAP Specifications*. Available from: http://www.w3.org/TR/soap/.

40. *Web Service Definition Language (WSDL)*. Available from: http://www.w3.org/TR/wsdl.

41. *UDDI Version 3.0.2*. Available from: http://uddi.org/pubs/uddi_v3.htm.

42. Schall, D., M. Aiello, and S. Dustdar, *Web Services on Embedded Devices*. International Journal of Web Information Systems, 2006. **2**(1): p. 5.

43. Dangoor, K. *TurboGears: Front-to-Back Web Development*. [cited 2009 May 20]; Available from: http://www.turbogears.org/.

44. Dangoor, K. *tgws - Google Code*. [cited 2009 May 20]; Available from: http://code.google.com/p/tgws/.

45. Chiu, K., M. Govindaraju, and R. Bramley. *Investigating the Limits of SOAP Performance for Scientific Computing*. in *High Performance Distributed Computing*. 2002: IEEE Computer Society.

46. Davis, D. and M.P. Parashar, *Latency Performance of SOAP Impelementations*, in *Second IEEE International Symposium on Cluster Computing and the Grid*. 2002. p. 13.

47. Govindaraju, M., et al. *Toward Characterizing the Performance of SOAP Toolkits*. in *Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*. 2004.

48. Bray, T., et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. 2008 [cited 2009 May 10]; Available from: http://www.w3.org/TR/xml/.

49. *XINS - Open Source Web Services Framework*. [cited 2008 Nov 4]; Available from: http://xins.sourceforge.net/.

50. Goubard, A. *XINS User Guide*. [cited 2009 May 20]; Available from: http://xins.sourceforge.net/docs/index.html.

51. Dangoor, K. *tgws - Google Code*. 2008 [cited 2009 January 8]; Available from: http://code.google.com/p/tgws/.

# APPENDICES

## APPENDIX A: DATA OBJECT DETAILS

A set of data type objects was selected for the experiments conducted during the course of this research. Although multiple experiments were conducted using different languages, synergies existed between the object types as much as possible. This appendix identifies the objects used throughout the experiments.

The data type definitions that follow were defined in the web service architecture implemented on the embedded device. These implementations are selected for this appendix as they identify the full gamut of data types that were tested throughout the research.

Due to document space configurations, the complex data types implemented using multiple layers of the same object will not be provided in this appendix. They are simply complex objects with an array of strings and a reference to a similar object. Due to issues found during the process of implementing these objects and the requirements of the web service framework used, the object could not be referenced recursively, as could be done using the Microsoft .Net Framework.

```
class SInteger(object):
   simpleInteger = int

   def RandomInt(self):
      self.simpleInteger = random.randint(1000,9999)
```

**Figure 51 – Integer Object**

```
class SString(object):
   simpleString = ''

   def RandomString(self):
      self.simpleString = ''
      for i in range(10):
         self.simpleString += string.ascii_uppercase[(random.randint(0,25))]
```

**Figure 52 – String Object**

```
class SIntAndString(object):
   simpleInteger = int
   simpleString = ''

   def Randomize(self):
      self.RandomInt()
      self.RandomString()

   def RandomInt(self):
      self.simpleInteger = random.randint(1000,9999)

   def RandomString(self):
      self.simpleString = ''
      for i in range(10):
         self.simpleString += string.ascii_uppercase[(random.randint(0,25))]
```

**Figure 53 – String and Integer Object**

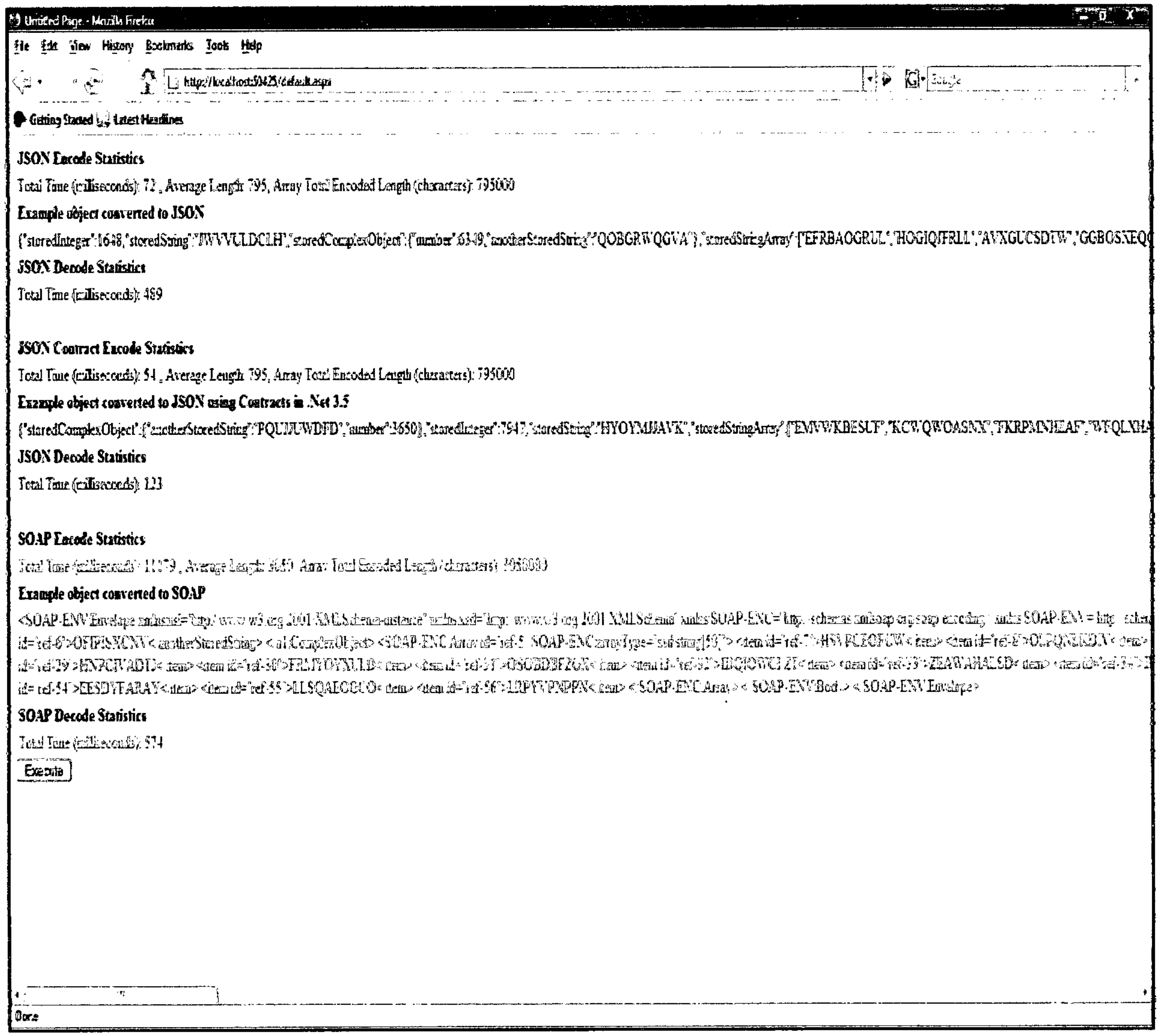
```
class ComplexObject(object):
   simpleInteger = int
   simpleString = ''
   complexObject = SIntAndString()

   def Randomize(self):
      self.RandomInt()
      self.RandomString()
      self.complexObject.Randomize()

   def RandomInt(self):
      self.simpleInteger = random.randint(1000,9999)

   def RandomString(self):
      self.simpleString = ''
      for i in range(10):
         self.simpleString += string.ascii_uppercase[(random.randint(0,25))]
```

**Figure 54 – Complex Object**

```
class ComplexObjectStrArray(object):
   simpleInteger = int
   simpleString = ''
   simpleStringArray = [str]
   complexObject = SIntAndString()

   def Randomize(self):
      self.simpleStringArray = [str]
      self.RandomInt()
      self.RandomString()
      self.complexObject.Randomize()
      self.RandomStringArray()

   def RandomInt(self):
      self.simpleInteger = random.randint(1000,9999)

   def RandomString(self):
      self.simpleString = ''
      for i in range(10):
         self.simpleString += string.ascii_uppercase[(random.randint(0,25))]

   def RandomStringArray(self):
      del self.simpleStringArray[0]
      for i in range(50):
         tempStr = ''
         for j in range(10):
            tempStr += string.ascii_uppercase[(random.randint(0,25))]

         self.simpleStringArray.append(tempStr)
```

**Figure 55 – Complex Object With Array**

## APPENDIX B: EXPERIMENT 1 DASHBOARD



JSON Encode Statistics

Total Time (milliseconds): 72 , Average Length 795, Array Total Encoded Length (characters): 795000

Example object converted to JSON

```
{"storedInteger":1648,"storedString":"FWVVULDCLH","storedComplexObject":{"number":6349,"anotherStoredString":"QOBGRWQGVA"},"storedStringArray":["EFRBAOGRUL","HOGIQJFRLL","AVXGUCSDTW","GGBOSXEQ
```

JSON Decode Statistics

Total Time (milliseconds): 489

JSON Contract Encode Statistics

Total Time (milliseconds): 54 , Average Length 795, Array Total Encoded Length (characters): 795000

Example object converted to JSON using Contracts in .Net 3.5

```
{"storedComplexObject":{"anotherStoredString":"PQUIJUWDFD","number":3650},"storedInteger":7947,"storedString":"HYOYMJJAVK","storedStringArray":["EMVWKBESLF","KCWQWOASNX","FKRPMNHEAF","WFQLXHA
```

JSON Decode Statistics

Total Time (milliseconds): 123

SOAP Encode Statistics

Total Time (milliseconds): 11079 , Average Length: 3650 Array Total Encoded Length (characters): 3650000

Example object converted to SOAP

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-ENV="http://schem
id="ref-6">OFIPISXCNV</anotherStoredString></a1:ComplexObject> <SOAP-ENC:Array id="ref-5" SOAP-ENC:arrayType="xsd:string[50]"> <item id="ref-7">HSVPCEOFCW</item> <item id="ref-8">OLPQNEKDJN</item>
id="ref-29">HNPCIVADIJ</item> <item id="ref-30">FRLIYOYNULD</item> <item id="ref-31">OSOBDBFZGX</item> <item id="ref-32">BIQIOWCJZI</item> <item id="ref-33">ZEAWAHALSD</item> <item id="ref-34">I
id="ref-54">EESDYFARAY</item> <item id="ref-55">LLSQAECGUO</item> <item id="ref-56">LRPYVPNPPN</item> </SOAP-ENC:Array> </SOAP-ENV:Body> </SOAP-ENV:Envelope>
```

SOAP Decode Statistics

Total Time (milliseconds): 574

Untitled Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:50425/default.aspx
Getting Started Latest Headlines
Execute
Done

Figure 56 – Experiment 1 Execution Dashboard

## APPENDIX C: TS-7800

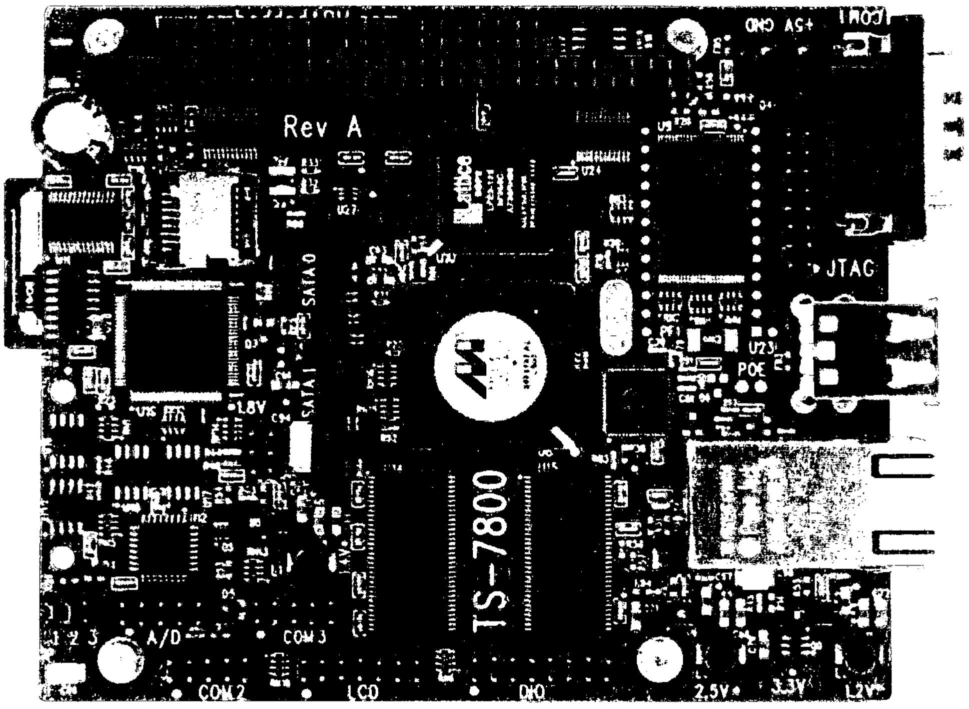


**Figure 57 – Technologic Systems TS-7800 Embedded Device [5]**