Deeply Embedded XML Communication – Towards an Interoperable and Seamless World

Johannes Helander
Microsoft Research
One Microsoft Way, Redmond, WA 98052, USA
+1-425-882-8080
jvh@microsoft.com

ABSTRACT

Current consumer electronics devices do not interoperate and are hard to use. Devices use proprietary, device-specific and inflexible protocols. Resources across device classes, such as personal computers and home appliances cannot be taken advantage of. Even recent efforts to connect sensors into networks concentrate on new, ad-hoc protocols that segregate the low-cost devices into their own little world.

If all classes of devices could speak the same language, they could talk directly to each other in ways natural to the application without artificial technical barriers. This would allow easily creating seamless applications that aggregate the capabilities of all the electronics. The interoperation adds value to all the devices.

Extensible Markup Language (XML) Web Services were conceived to solve the e-business interoperation problem. After decades of failed attempts with EDI, SNA, DCOM, CORBA, and other similar technologies, XML and its communication specification SOAP has proven itself to be a viable technology. If XML is good for e-business, could it also be good for embedded systems communication?

This paper argues that XML and SOAP indeed can be useful in small devices. Solutions to performance questions are available and techniques are outlined here. New unique challenges, such as heterogeneous configuration, privacy and security issues, and real-time requirements (e.g. for gaming) are identified and solutions outlined. A prototype implementation for low-cost microcontrollers is described with numbers included.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]:
Real-time and embedded systems

C.2.1 [**Network architecture and design**]:
Wireless communication; Network communication

General Terms

Algorithms, Design, Performance, Security, Standardization

Keywords

Invisible Computing, XML, SOAP, Service Oriented Architecture, Home Networking, Embedded Systems

EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.
Copyright 2005 ACM 1-59593-091-4/05/0009...$5.00.

1. INTRODUCTION

Rice and seaweed can be eaten separately. However, when put together they taste a whole lot better. This *sushi-effect* is a case where putting two things together adds value to both. In networking this is more commonly known as *Metcalfe's Law*, after the inventor of the Ethernet. Another way to put this is to state that it is counter-productive to separate different device classes into separate categories. Each category will then have to build critical mass for joint utility separately. If instead existing computing resources are taken advantage of, their momentum and critical mass can aid in pushing the information revolution into new areas.

There are two converging technology trends that are in the position to create a breakthrough in consumer electronics—to create a seamless and interoperable world.

1. Internet standards are no longer limited to moving data from one computer to another. Instead common representation of data and distributed computing is emerging from Service Oriented Architectures (SOA) and XML. This is essentially a standardization of the presentation and application layers, or levels 6 and 7 in the old ISO/OSI model. The leading efforts are in the two layers: SOAP (Simple Object Access Protocol, [1]) defines how to use XML for communication and WS-* (WS-Security, WS-Transfer, WS-Management, etc., [2]) specifications define how to use SOAP for specific purposes.
2. Sufficiently powerful 32-bit microcontrollers are becoming cheap, small, and energy-efficient. It is already feasible to put these single-chip microcontrollers into almost any consumer application and there is decreasingly any need to aim for the very lowest end, the more slowly evolving 8-bit microcontrollers or analog circuits to control devices. This means that more intelligent software can be used, even while performance is still of essence.

When put together it makes sense to ask whether there is any need for category specific communication protocols anymore, or would it be feasible to use the same protocols and data representations on all device categories. The answer appears to depend on: A) Can XML perform well enough? B) What necessary features are the XML standards not addressing? C) Are the new microcontrollers cheap and good enough?

The rest of this paper is organized as follows: Section 2 outlines some XML usage patterns in the embedded space; sections 3 and 4 outline solutions to the performance question; sections 5, 6, and 7 address new features; sections 8 and 9 provide "proof by construction" and section 10 draws conclusions.

What about the microcontrollers? The market will tell but it looks good. At the time of writing this paper multiple manufacturers produce very low-power ARM single chip computers that are sufficiently powerful to run the software described in this paper in the $5 price range. The prices are dropping quickly and the use of the new microcontrollers in devices is becoming easier.

2. HOW XML IS USED IN DEVICES

XML is a semi-human readable stream of data, where each data item is tagged with a name. The formats of messages and the names of the tags are specified in a *schema*, which itself is expressed in XML. One document can be described by multiple schemas and the tags in the document have namespaces that facilitate resolving what tag belongs to what schema. The result is a structured document that is mostly self-descriptive. As its name indicates, XML is also extensible. New tags and schemas can be introduced and mixed with old documents without changing the meaning of the old tags. In SOAP specifically, the extension feature is additionally controlled by an explicit *mustUnderstand* attribute allowing some control over what extensions can safely be ignored in previous implementations. These properties make XML useful in heterogeneous environments where different computers use different software and different versions.

While XML and SOAP specify message formats and how the messages should be processed, it leaves the implementation open. Some implementations may understand everything, where others only understand a few predefined messages.

Messages are sent to objects in servers. A server typically sends a reply message but that depends on the message specification and schema. Unidirectional messages are traditionally called *asynchronous* and can be used to form arbitrary message patterns. Since everything that receives a message is a server, this is sometimes called a Service Oriented Architecture. Messages sent to objects fall largely into one of two categories: method calls, such as *RobotMoveArm* or property sets, e.g. *Get SensorState.* The choice between the styles is up to the specific use depending whether they deal with active or passive objects. The property set approach is facilitated by standard methods in WS-Transfer, WS-Enumeration, and WS-Eventing, while the actual data format is specified in an application specific schema. In the method approach the messages themselves are specified in the application schema. Note that in current implementations some elements are specified in a separate WSDL document rather than the regular schema document, but essentially this is just a schema extension.

For instance, a sensor device could provide its sensing and calibration parameters through one property set and the reading values through another. The user of the sensor values would then subscribe to the values using WS-Eventing. All that is needed to represent the sensor as a SOAP service is to define the schema and map the actual hardware values to the property set structure. If the sensor device also had an actuator, it could be controlled through appropriate methods, such as

<MoveSteps>5</MoveSteps>.

Since XML is somewhat human readable, it can also be used to represent domain specific programming languages and message pattern specifications, without the need for new parsers. One example of this is presented in section 7. In a way it is back to the future, when LISP used to have the same representation for programs and data.

XML messages can be exchanged for control and discovery messages. XML can also be used for payload data, streaming, data-flow type programs, etc. The same messaging infrastructure can be used from the huge data center machine to the tiny deeply embedded device.

One example of a practical interoperation problem is remote management of computers. The computers might be in different locations and have different operating systems and hardware. They also can be at different points in the host OS lifecycle, such as pre-boot, OS running and post-run. WS-Management is an industry standard that replaces complex vendor specific mechanisms with a coherent SOAP interface. It uses a property sets to access objects like processes, disks, and users. It uses methods to expose object specific functionality, e.g. PhysicalDisk::Format or ComputerSystem::Reboot. The WS-Management interface is implemented, amongst others, by the latest Windows® operating system. A service processor, such as a smart network card or a chassis manager, running WS-Management would be ideal for installation, monitoring and repair. Interestingly this calls for an embedded low-cost SOAP implementation even for managing big computers due to the management controllers' cost structure. Results from a preliminary implementation are included in section 9.

3. PERFORMANCE

Since the first XML Web Services implementations were geared towards high-end computers and were not well optimized, a common belief emerged: XML would be too heavy for low-cost devices. But is this really the case? Implementation cost can be evaluated through an alternative, light-weight implementation (section 8). So what are the fundamental costs and can they be mitigated sufficiently?

1. Silicon—footprint: The footprint needs to be small enough so that the cost, size, and energy consumption of the microcontroller stay low enough.
2. Bandwidth—size of messages: In a slow network or a low-power wireless, transmitting overly large messages takes time, making operations slow, and prevents other use of the network.
3. Energy—parsing overhead: The overhead of parsing, interpreting, and creating messages takes CPU cycles, where every cycle consumes a bit of energy, draining a battery. Transmitting and receiving data also consumes energy proportionally to the size of the message.

(1) Footprint is largely an implementation issue. If a small amount of code can do all the work, the code size and consequently ROM footprint will stay small. Our solution here is a table driven serializer. Rather than having specific code for different messages, a tightly written interpreter matches messages with a compact metadata table that describes the messages. The results presented in section 9 show that this can and has been achieved.

RAM footprint is dominated by network and message buffers and program stacks. In addition, it is important that as much data as possible is put into ROM as RAM is much more expensive. For this reason the above mentioned metadata table is placed in ROM –while if extended from XML schemas at run time the extensions go into RAM. Stack space is controlled by limiting the number of threads that do processing; by keeping the code straightforward by avoiding excessive recursion; and by delaying stack allocation until actually needed. An event driven serializer allows using the same thread to process multiple messages and processing each one while it is being received. The message is processed directly into its final form without intermediate object trees. A zero-copy networking interface allows processing the messages directly to and from the same memory buffers where the data was initially received without having to copy into secondary buffers. This saves both energy and space and applies to all stages, including encryption. Finally, messages are processed into continuations that are like threads except stack allocation is deferred until actual execution time.

(2) XML tends to be verbose, with long strings naming simple things. Moreover since messages try to be self-descriptive and stateless, a lot of mostly constant data is repeated from one message to another.

In many cases the verbosity does not matter and there are well established ways of using binary attachments for bulk data (e.g. video frames). The additional overhead in a message is still small compared to all the other networking layers and even the somewhat bloated messages are still small enough. However, in a few cases the overhead is simply too great—this can happen with e.g. certain encodings in WS-Security, where message sizes explode and cannot be compressed. Those encodings are simply unsuitable for embedded use. In other cases, however, a simple compression can take care of the bloat.

(3) Energy is consumed in four significant ways related to messaging: executing, idling, transmitting, and receiving.

Every instruction executed consumes a bit of energy but added together it matters little how fast the instructions are executed so a 32 bit microcontroller is about the same as an 8 bit microcontroller in this regard. The amount of processing is affected by the data representation. Due to its semi-human readable representation XML implies conversions from internal computer data representations to textual representations. For instance, converting a number from binary to textual representation requires a large number of divisions, only to be converted back at the other end. Some of the overhead can be mitigated by an efficient implementation. Another approach is to compress the messages in such a way that the compressed message can be processed directly. Section 4 below discusses how to do that.

Idle time power consumption is affected by the leakage current of the microcontroller. The leakage depends on the transistor count and is somewhat higher on a 32 chip part than an 8 bit chip. One solution is to shut off the power supply to the chip completely, leaving it to an auxiliary circuit to turn it back on later. This was technique was used in the Microsoft smart watch. Transitions between idle and running can also be high so it is desirable to do as much useful work as possible once there is something to do.

Transmission energy is a simple function of message size. Receiving instead takes power whether anything is sent or not. In low-power radios receiving can take as much or more power than transmitting; thus it is important to be able to turn the receiver off when there is no data to receive. The complication is that without the receiver on there is no way of detecting when somebody is sending so advance coordination is required. Section 7 outlines one approach to that.

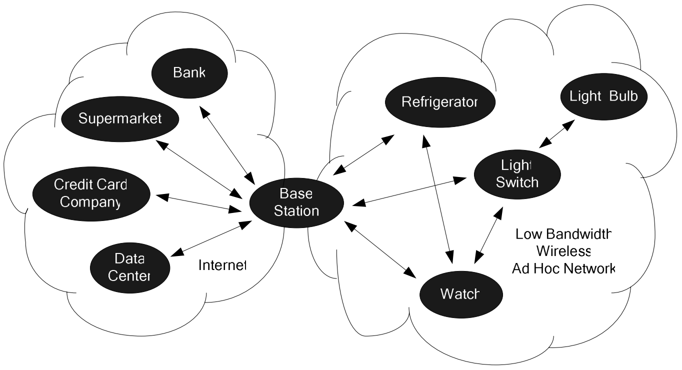
4. COMPRESSION

When data is encoded as XML it presents an infoset defined by its schema and has a canonical representation. If the XML message is first generated and then compressed, say using zip, it is still XML and has the same properties while it is no longer readable since the process can be reversed by decompressing the data. If the compressed form could be generated directly, without first generating the textual representation, the data would consequently still be XML as the textual representation can be mechanically generated from the compressed document. This would be achieved while avoiding the performance hit from generating the intermediate textual representation. The challenge is thus to compress the data in such a way that it is efficient to generate directly from native data on a given machine. The author proposes a *template based compressor* where partially evaluated XML message templates can be referenced and fully instantiated with filling in the unknown parts—this is essentially a macro facility that produces the XML parsing events when evaluated. From this it is possible to either generate the canonical textual XML or directly the native data.

Some templates can be automatically generated from a schema, e.g. corresponding to a method call or structure. Others can be generated at runtime and transmitted thus facilitating delta encoding. Furthermore, a given schema can be compiled into a pre-tokenized form, where numerical values are assigned to known element names and strings (QNames). By encoding a message in the pre-tokenized template form the message size can be reduced to only a template index and parameter values. Say an *Add* method call that in its textual representation is some 150 bytes can be encoded to three bytes: One byte identifies that this is an add, and one byte each is the value of each addend. The cost here is that there is some connection state that needs to be negotiated, namely the mapping of indexes and known templates needs to be agreed on but this can also be by reference: point to an external location (by Uniform Resource Identifier or URI) that describes the encoding.

The implementation of the serializer uses the same code to deal with either textual or compressed messages, showing that the compression feature can come at little overhead.

While the author envisions the main use of the compression feature to be optimizing transmission and processing, it is also noted that some very small devices might only implement a small number of predefined compressed templates. This would allow essentially raw binary data to be mechanically treated as XML messages by other parties—given an extra discovery step that allows the client to resolve the templates the device knows. This can potentially be a solution to using XML Web Services on microcontrollers that are too weak to run even the very compact implementation stack presented in section 8.



5. PRIVACY AND SECURITY

While efficient implementations and compressed representations can make the web services perform on the tiniest computers, there will be some new challenges not yet addressed by the "big" web services. Those arise from where microcontrollers are used: interfacing with the real world and with humans living in the real world.

Automating a home with smart devices and creating interoperating entertainment as well as medical electronics hold great promise in enabling elder people to live longer in their homes and making the lives more comfortable to all of us. But installing communicating sensor devices in a home is also a risk.

The devices are essentially surveillance equipment. Who will want to pay for surveillance on themselves? It seems that to create a viable market there must be strong assurances that the information is tightly controlled by the inhabitants of the home. Devices that control the physical world are similarly risky: Doors could be unlocked, stoves turned on, and health devices follow the wrong protocol if the control falls in the wrong hands. It is thus essential to have strong communications and data security. Since it is also essential that a home can be setup independently without an active network connection and that devices will work without outside facilities, an independent trust model is called for.

The author shows in [1] how to use public key encryption and the resurrecting duckling protocol [4] to author independent trust domains and arrange key exchange. See also the text box below.

Unlike WS-Security, that encrypts pieces of XML and tends to generate extremely large messages, the embedded web services encrypt entire messages. This means the content can be compressed before encryption and also requires less processing.

The independently created trust domains (i.e. the family certified by the same mother device) can be federated using standard public key cryptography and simple trust exchange (e.g. write hash of public key on check, when the check clears, the bank trusts you). The federation enables internet scale trust, e.g. for e-business, while allowing completely independent home setup without external certification authorities or other connectivity.

6. INTUITIVE SETUP

The touch based trust setup has an interesting side effect: when multiple devices are touched and the time along with other known state (location, temperature, etc) is recorded, a context history is created. This context history can be data mined for possible human intent—a clustering algorithm is presented in [5]. For example first touching a light switch and then a torchier intuitively indicates that the light switch should control the torchier light. The context history analyzer makes the computer think the same.

A smart home setup thus proceeds as follows: 1) bring one or more new devices home; 2) touch each one once with your watch; 3) the home is now ready to go. The same simple interaction is used both to establish trust and security and to assign functional associations between the devices. The result of the heuristic is displayed by flashing LEDs on the devices, the light itself, or in other available ways. If the human is unhappy with the result or wants to change an old association, she can touch a new pairing to create a new association. No archaic menus or setup scripts are required.

```
<behavior name="PlaySong">
  <action name="RemoteControl"
          endpoint="node:instigator/music.cob">
    <message destination="MusicProducer/*"/>
  </action>
  <action name="MusicProducer"
          endpoint="node:cdplayer/music.cob"/>
    <repeat count="10000" Period="P0.1S"/>
    <message destination="MusicConsumer"/>
  </action>
  <action name="MusicConsumer"
          endpoint="node:speaker/music.cob"
          tolerance=”P0.0001S”/>
    <repeat count="10000" period=”P0.1S”/>
   </action>
   <sampling destination="node:instigator"
             interval="20" number="2"/>
</behavior>
```

7. REAL-TIME

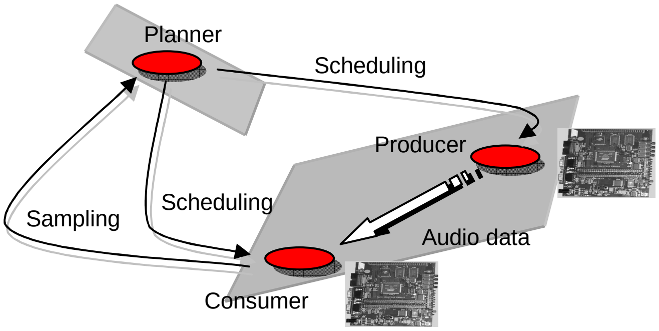
Interacting with the real world implies interacting in real time. This means that the computers not only need to do the right thing but also do it at the right time. Adding temporal predictability in a heterogeneous distributed system, such as the one formed by interoperating devices, is not well addressed by traditional real-time methods. Traditionally industrial processes, airplanes, and other real-time systems were analyzed in minute detail and all possible interactions enumerated. In the new environment where new devices can be introduced arbitrarily and the devices are all different the static analysis breaks down. Instead a self-adaptive approach is called for.

A behavior pattern (see box above) describes a distributed activity in an abstract way in a domain specific language. This is expressed in XML so the same parser can be used. When a human presses “Play” on the remote control, this is mapped to a {when, what, how, importance} tuple, where *how* is the pattern, when is 0.1 seconds from now, and what is the song “Yesterday”. The importance is expressed as a probability of success, a confidence.

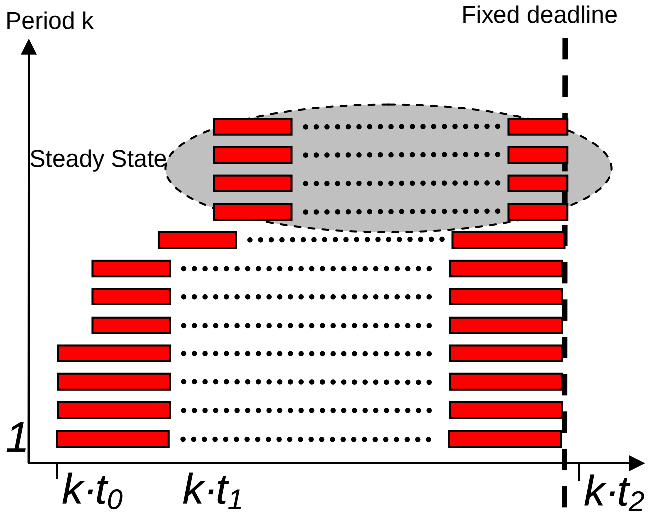
The instructions are fed into a planner software component. The planner instantiates the pattern spatially and temporally. The spatial instantiation is a normal device discovery that determines which devices are needed to do the work. The temporal instantiation determines what resources are needed when and for how long. The planner predicts resource needs based on performance monitoring, similar to industrial quality control, with sampling schedules and stochastic modeling.

The Resurrecting Duckling Protocol in a Smart Home

1. A new device, e.g. a light switch, is purchased and turned on.
2. On the way home an accelerometer is used to generate a new random RSA key pair using ambient vibrations.
3. One device, e.g. a watch is picked to be the *mother.* It signs its own public key.
4. The light switch touches the watch. Since the switch just came out of the egg it believes the first device it touches is its mother. The watch sends its own certificate and the switch sends its public key over a touch based channel (electrostatic, short range radio, USB).
5. The mother signs the switch’s public key and sends the certificate over a public wireless link. The switch is now part of the family.
6. When two members of the device family want to talk to each other for the first time, they exchange their certificates over the public radio and verify them using the mother’s certificate.
7. A symmetric key encrypted by RSA is exchanged and cached on each device.
8. The symmetric key is used to encrypt all the data exchange between the two devices using AES ensuring private trusted communication. Cryptographic hashing is used to ensure message integrity of large messages.
9. If and when the switch is sold in a garage sale, the watch can instruct it to crawl back into the egg and invalidate its certificate.



The instantiated pattern is a specific schedule, a task for each participating device. The planner sends the task as XML over to the worker node, which next compares it to its prior commitments and answers either *yes* or *no.* If all the workers are affirmative, the planner sends a *go* message and the audio streaming is started. The time consumed at each stage of the processing is recorded and measured. Samples of the measurements are sent back to the planner, which uses the feedback to adapt the schedules. The adaptation is driven by the samples: the stochastic model is updated and more accurate predictions are produced. One stochastic process is presented in [6]. A possible adaptation is visualized below.



It is interesting to note that the stochastic approach allows running distributed tasks on platforms with variable levels of real-timeliness. The predictability of a given platform is determined by the variance in the observed time to process tasks and is reflected in the amount of time needed to achieve a given level of confidence.

Another interesting consequence of pre-declaring task schedules across multiple machines is that a receiver of messages knows when to expect (within tolerances) a message. It consequently knows when it is *not expecting* messages, and can turn off its radio receiver during that time enabling potentially significant power savings.

8. IMPLEMENTATION

How can we be sure XML Web Services actually work on a microcontroller? The only real proof is to implement a stack. Our implementation follows a component design. The base component is a small real-time operating system that supports the other components and includes a low-level constraint based scheduler (an extension of earliest deadline first, see [7] for an explanation).

The RTOS supports a TCP/IP stack, an XML processor, cryptographic primitives, various service handlers and applications, and a table driven serialization engine that matches the parsed XML with native data representations (stacks and structs) to each other based on a metadata table that is compiled from interface description schemas.

The distributed real-time scheduler builds on top of the base constraint scheduler and adds admission control and scheduling of tentative, repetitive, and inter-dependent work items.

The component design allows a pay-as-you-go approach to composition. While interfaces are reused and the same code can be utilized by multiple components, there is no general purpose compromise. Instead precisely the features required by the given application are included. This means that features need to be selectively used in low-budget hardware.

The entire system is written in the C language and works on multiple processors and microcontrollers. The following section presents specific numbers for one configuration on one popular microcontroller.

9. RESULTS

The performance was evaluated on an ARM7 microcontroller development board [8]. We observe that the secure XML software can run on a computer that has 256KB ROM and 32KB RAM. This amount of memory is available on many modern microcontrollers of interest.

Files	ROM	Static RAM	Heap RAM	Stack RAM	Total RAM
BASE	24,676	1,940	2,837		2,777
DRIVERS	11,464	332	896	2,288	3,516
TCP/IP	77,024	3,424	2,648	3,400	9,472
XML	7,860	16	88		104
SOAP	29,504	280	996	4,320	5,596
SECProto	14,180	604	1,848	2,648	5,100
AES	16,532	8			8
RSA	9,784	28	24		52
SHA1	5,436	8			8
C-Library	7,620	12			12
TOTAL	**204,080**	6,652	9,337	12,656	**28,645**

Footprint (arm - in bytes) at peak usage

We evaluate whether the solid cryptography is feasible on low-cost devices. The table below reveals that the two significant costs are key generation and RSA private key operations. The former only needs to be done once, and can be primed on the way home. RSA private key operations are needed for certificate signing and key exchange. Each need to be done only once but cannot be done before the device was touched. Luckily the certificate does not have to be signed while touching so the interaction itself is quick.

The working of the stochastic planner was estimated through sampling. A simple test method does 20000 multiplications. Starting with no information the planner uses an application provided guess.

Algorithm	Operation		Latency on a 25 MHz ARM 7		
			Average	Standard deviation	Per KB
1024-bit RSA	Generate a key pair		290 s	56%	N/A
	Private key	Encrypt/decrypt a block (128 bytes)	12.9 s	<1%	103 s
	Public key	Encrypt/decrypt a block (128 bytes)	0.667 s	<1%	5.34 s
128-bit AES	Encrypt/decrypt a block (16 bytes)		0.254 ms	<1%	16.3 ms
SHA1-HMAC	1024 bytes		79.6 ms	<1%	79.6 ms

Speed of cryptographic primitives

Once the planner gets real samples it uses them with smoothing between each step. The calculation times include formatting and sending the reply message. The table below has the numbers. The estimate is produced by the live planner, while the mean and deviation have been calculated offline for reference from the raw measurements.

Step	Estimate 95% conf	Measured mean	Standard deviation	Confidence	
				95%	99%
1	339	337	1.7%	1.0	1.4
2	341	337	1.6%	1.0	1.4
3	346	337	1.8%	1.0	1.4

Time measurement and prediction of a CPU intensive task – times in milliseconds, 32 samples per iteration on embedded microcontroller board. The confidence number indicates the extra time allocated for jitter. Fixed point integer arithmetic rounds the number up slightly.

Since the low-level RTOS scheduler did not produce much jitter, the test was also executed on a PC running WindowsXP with the SOAP middleware stack on top. Running without an underlying real-time scheduler introduces more uncertainty but the planner still deals with it correctly and produces a larger confidence allocation to cope with the increased jitter. As the CPU is faster a million multiplications is done each time. From a steady state the number of calculations is dropped to half. The table below shows how the planner adapts to the drop. The planner adapts to the larger jitter by padding the estimates.

Step	Estimate 99% conf	Measured mean	Standard deviation	Confidence 95%	
					99%
1	126	123	6.4%	1.9	2.5
2	124	120	14%	4.2	5.5
3	69	55	2.1%	2.8	3.7
4	58	55	2.9%	3.9	5.2

Figure 7, Time measurement on PC in milliseconds. After the steady state at step 2, the workload is cut in half and the estimate adapts to the new load.

Since the main point of XML is interoperation, we need to evaluate how well that promise is realized. The SOAP stack presented in this paper recently participated in a WS-Management interoperation workshop, with a dozen implementations from several major corporations. The embedded web service stack easily passed the interoperation test with only small problems with a couple of implementations due to it not supporting HTTP chunking on the client end. The stack has also proven to interoperate with commercial SOAP stacks with other load. It is easy to add new web methods and properties.

10. CONCLUSION

This paper argued that XML Web Services are desirable and feasible for embedded systems use. It described a number of techniques in how performance can match the strict resource constraints in modern microcontroller systems and how XML can be used to address challenges in interoperation, ease of use, security, and real-time. A prototype implementation offered proof by construction that using XML indeed makes sense. The result is a small, cost-effective implementation that supports a number of standard specifications, such as WS-Management.

While XML and SOAP are useful in embedded systems, their specific needs require special consideration and not all WS-* standards can be adopted as is. Further work is required to create standards addressing the small systems requirements and tune existing standards. Performance refinements will further allow extending the scope of the seamless world. Finally, new interesting applications are needed to drive adoption.

11. REFERENCES

1. SOAP Version 1.2 Part 1: Messaging Framework—W3C® Recommendation 24 June 2003, http://www.w3.org/TR/soap12-part1/
2. Web Services Architecture—W3C® Working Draft 8 August 2003, http://www.w3.org/TR/ws-arch/
3. Helander, J. and Xiong, Y.: Secure Web Services for Low-cost Devices, *8th IEEE International Symposium on Object-oriented Real-time distributed Computing,* Seattle, May 2005.
4. Stajano, F. and Anderson, R. The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks *LNCS 1796*, Springer-Verlag, 1999.
5. Helander, J.: Exploiting Context Histories in Setting up an e-Home, *First International Workshop on Exploiting Context Histories in Smart Environments*, Munich, Germany, 2005.
6. Helander, J. and Sigurdsson S.: Self-Tuning Planned Actions: Time to Make Real-Time SOAP Real, *8th IEEE International Symposium on Object-oriented Real-time distributed Computing*, Seattle, May 2005.
7. Jones, M. B., Roçu, D., Roçu, M.: CPU Reservations and Time Constraints: Efficient Predictable Scheduling of Independent Activities, in *Symposium of Operating System Principles,* St-Malo, France, 1997.
8. AT91M63200 Summary, AT91 ARM Thumb MCU, http://www.atmel.com/dyn/resources/prod_documents/1028S.PDF
9. The embedded web services implementation is available at http://research.microsoft.com/invisible/