BILINGUAL PUBLISHING CO.
Pioneer of Global Academics Since 1984

Journal of Computer Science Research
http://ojs.bilpublishing.com/index.php/jcsr

ARTICLE

Energy-Efficient Transaction Serialization for IoT Devices

Daniel Evans*

Pace University, United States

ARTICLE INFO

Article history
Received: 6 January 2020
Accepted: 20 January 2020
Published Online: 31 May 2020

Keywords:
Energy efficiency
Data serialization
IoT serialization
XML equivalence
JSON equivalence

ABSTRACT

This article presents two designs, the Transaction Serial Format (TSF) and the Transaction Array Model (TAM). Together, they provide full, efficient, transaction serialization facilities for devices with limited onboard energy, such as those in an Internet of Things (IoT) network. TSF provides a compact, non-parsed, format that requires minimal processing for transaction deserialization. TAM provides an internal data structure that needs minimal dynamic storage and directly uses the elements of TSF. The simple lexical units of TSF do not require parsing. The lexical units contain enough information to allocate the internal TAM data structure efficiently. TSF generality is equivalent to XML and JSON. TSF represents any XML document or JSON object without loss of information, including whitespace. The XML equivalence provides a foundation for the performance comparisons. A performance comparison of a C reference implementation of TSF and TAM to the popular Expat XML library, also written in C, shows that TSF reduces deserialization processor time by more than 80%.

1. Introduction

The rapid development of the Internet of Things (IoT) has renewed interest in transaction processing efficiency. The paper "Energy Efficiency: A New Concern for Application Software Developers" [1] recently detailed the interaction between energy issues and software in IoT and mobile devices: "… wasteful, poorly optimized software can deplete a device's battery much faster than necessary." The initial energy supply of many IoT sensor devices limits their deployment lifetime. These devices sense information and transmit the results of sensing. They receive instruction from remote controllers. They continually serialize and deserialize transactions to and from a communications medium. Any reduction in processor time used by transaction serialization/deserialization contributes to an increase of the deployed lifetime of an IoT device. This work describes the Transaction Serial Format (TSF), whose primary goal is to use as little energy as possible to perform the serialization/deserialization tasks. The Transaction Array Model (TAM) supports the efficiency of TSF. Although a dynamic data structure, TAM directly integrates information from the TSF format.

1.1 Serialization Overview

The need for serialization and deserialization, converting data to and from serial media such as communications networks and long-term storage devices, has spawned many designs. Wikipedia lists 35 different formats in its "Comparison of Data Serialization Formats" page. The differences are many and the reasons for development of each are not always clear. However, they have a common

**Corresponding Author:*
Daniel Evans,
Pace University, United States;
Email: de36804p@pace.edu

 DOI: https://doi.org/10.30564/jcsr.v2i2.1620

feature: the need to preserve data structure. The data structuring facilities of random access memory are not feasible when accessing data sequentially from beginning to end one byte at a time.

Serialization formats generally fall into two categories: formats specific to an application, and general formats intended for use by any application. Examples of application formats are Apache Avro [2] and Apache Parquet [3] both designed for the Hadoop parallel processing system. Binary JSON (BSON) [4] adds binary representations and extensibility to JSON-structured data in the MongoDB database system. Also in this category are language-specific serializations such as Java Object Serialization [5], Python Pickle [6], and Perl's DataDumper module [7], the first of many PERL serialization modules. Remote Procedure Invocation produced a number of formats designed to marshal parameters for a remote procedure call and to return the results. The Common Object Request Broker Architecture (CORBA) specifies the InterORB Protocol [8] for communication between clients and object request brokers. The Java library provides the Remote Method Invocation (Java RMI) [9] for Java programs to invoke remote methods available on any machine running an RMI server. D-Bus [10] is designed for both interprocess control on a local machine and remote invocation. Apache Thrift [11] is a design for cross-platform RPC. XML-based RPC serialization came from Microsoft in the form of XML-RPC [12], the ancestor to the World Wide Web Consortium (W3C) SOAP standard [13].

General serialization formats are for any application, and TSF is in this category. In addition to the basic requirement of preserving data structure, many formats incorporate additional features that:

(1) Reduce the size of serialized data to minimize data transmitted

(2) Use an external schema to describe the serial format(s)

(3) Work with multiple character encodings, such as UTF-8, UCS-2, and UCS-4

(4) Use only characters recognized by a text editor so that the serialized data is human-readable or easily changed

(5) Address some additional need, such as machine-independent data definitions that could be exchanged by machines of different architecture, or the restriction to 7-bit or 8-bit clean so that data can be transmitted through gateways and across networks with differing characteristics

In the following, we examine serialization formats that incorporate one or more of these features, and note how TSF compares to them.

Data compaction was an early issue for the telecommunications industry when transmission speeds were much slower than they are today. The International Telecommunications Union (ITU), the standards body for international telecommunications, released in 1984, as part of CCITT-X.409, the Abstract Syntax Notation, version 1 (ASN.1), an interface description language (IDL). In 1988, ASN.1 became a separate standard, X.208 [14]. ASN.1 has gone through several revisions and is currently at Revision 5, but has not lost its original name, ASN.1.

ASN.1 is a declarative language (also called a schema language) for describing messages as general structures of arbitrary data types. A schema is a description of the serialization format, which both sender and receiver have. An ASN.1 schema declares data types first, and then declares messages as structured sequences of previously declared data types. ASN.1 provides an extensive set of built-in type constructors as a foundation for complex types. Every defined type has an unambiguous serial encoding, commonly known as type-length-value encoding. ASN.1's Basic Encoding Rules (BER) define the rules for serializing data using bit- and byte- aligned fields. Initial implementations of ASN.1 compiled a schema of data types and message declarations into source code in a language such a C, providing both encoding and decoding functions. The generated encoding and decoding functions, specific to the described messages, could in turn be included as source code, or compiled and linked as a library, into application programs that sent and received the messages. ASN.1 found wide use within the telecommunications industry. A subset of ASN.1 became the Simple Network Management Protocol's language for describing SNMP's MIBs (Management Information Base) [15], which are abstract descriptions of network devices subject to management by the protocol. ASN.1 also found use in applications that use some form of the X.500 series of standards from the CCITT, such as the exchange of cryptographic metadata with X.509 certificates [16].

ASN.1 uses type codes and a schema to define and preserve structure. Another common structure-preservation technique is the use of delimiters that signify the beginning and end of data and provide for data nesting. XML [17] and JSON [18], two of the most popular formats in use today, both use this technique. XML uses named parentheses, called open and close tags, to delimit and nest data. JSON uses two types of structure delimiters, the characters "["and"]" for arrays, and the characters "{"and"}" for collections, which are named sequences of data. Collections synonyms are associative arrays, maps, hashes, or dictionaries in other computer languages. The YAML format [19] uses indentation and the natural delimiting provided by line end characters to preserve data structuring.

DOI: https://doi.org/10.30564/jcsr.v2i2.1620

The lines in the YAML format have additional syntax, which may optionally include "flow" formats, close to the delimited design of JSON. YAML is also user editable with any general text editor program. Another delimited, editable format is "s-Expressions", originally designed by John McCarthy, the inventor of Lisp, and described in the Internet Memo [20] by Ronald Rivest. Like Lisp, it uses left and right parenthesis as delimiters and nests delimited data to provide structure. TSF does not use delimiters to define serialized structure, but takes an approach that is closer to ASN.1, although without a schema. Although editing is not a design goal, TSF is editable with a normal text editor, if done carefully to update lengths after adding or deleting characters.

The concern for compact representations appears in many early serialization formats, coincident with communications speeds slower than today's. ASN.1's Basic Encoding Rules are the prototypical example. Later formats also dealt with compact representations. XML representation efficiency became an issue soon after XML use became widespread. The W3C chartered the XML Binary Characterization (XBC) Working Group and the Efficient XML Interchange (EXI) Working Groups in 2004. Both groups worked in the area of efficient representation of XML. The XBCWG produced the first draft of "XML Binary Characterization Properties" [21] in late 2004. The EXIWG produced its first draft in mid-2007 [22].

The work of these groups has always been informed by the desire to preserve as much of the primary XML specification as possible, and to be aware of XML schema definitions when they exist.

"EXI is schema 'informed', meaning that it can utilize available schema information to improve compactness and performance, but does not depend on accurate, complete or current schemas to work."

The EXI working group ultimately produced a very large code implementation of the specification and made it publicly available, but it does not appear to have been widely used.[1]

In 2012, the W3C-chartered MicroXML Community Group produced the MicroXML specification [23]. The justification for MicroXML is at the beginning of the specification document.

"MicroXML is a subset of XML intended for use in contexts where full XML is, or is perceived to be, too large and complex. It has been designed to complement rather than replace XML, JSON and HTML. Like XML, it is a general format for making use of markup vocabularies rather than a specific markup vocabulary like HTML."

MicroXML simplified XML by, among other things, eliminating DOCTYPE's, namespaces, processing instructions, CDATA sections, and character set options. However, as MicroXML is a subset of XML, no alternate serialization format was proposed.

Improved line speeds resulted in less attention to compact representations, but the rise of mobile devices with their bandwidth limitations kept compact representations a concern, as evidenced by the recent Compact Binary Object Representation, described in 2013 in RFC 7049 as "… a data format whose design goals include the possibility of extremely small code size, fairly small message size, …" [24]. However, alternate compaction techniques have found wider commercial use. Web servers that download JSON have preprocessed the files to remove non-syntactic whitespace, and sometimes renaming all the variables to minimize their length, which has the effect of obscuring the source code. The files can also be processed by a standard compression algorithm, as all browsers have the ability to process several compression formats. In TSF, the elimination of redundancies yields some compaction, but compaction is not a primary motivation for the format. A TSF message is usually slightly smaller than its JSON equivalent. Any standard compression technique can further reduce the size.

Schema-based serialization formats tend to be more compact as they can eliminate some or all type information from the serialization by maintaining it in a separate schema, generally written in a custom interface description language (IDL). ASN.1 again is the prototypical example. A more recent example is Google's Protocol Buffers [25]. Once written, a Protobuf schema compiler generates source code for any supported language. The generated code then is included in any program that uses the serialization described by the schema. Flatbuffers [26], similar to Protobuf, can use both its own IDL and that of Protobuf. TSF is not a schema-based serialization, but it does use a technique called "zero-copy deserialization", also used by Flatbuffers, to reduce the number of memory allocations required to construct the internal representation of a deserialized object.

Differences in machine architecture have motivated some serial formats. The External Data Representation first defined in RFC 1832 in 1995 and subsequently updated eleven years later in RFC 4502 [27] provides serial representations for standard binary data types such as signed and unsigned integers, 64-bit integers, called hyper integers, floating point values, enumerations, fixed length arrays, and more. However, the popularity of character-based serializations seems to have provided an alternate, simpler way to handle architecture differences,

1 The entire OpenEXI package download is 355 Mb. Compare this to the total download size of 84Mb for the Expat XML parser.

 DOI: https://doi.org/10.30564/jcsr.v2i2.1620

as conversions from character forms to internal data types are available on any machine and in every language. TSF delegates the data format definitions to the application.

There are two serialization formats that have some coincidental similarity to TSF. Bencoding, part of the BitTorrent specification [28], serializes string data with a count followed by a delimiter preceding a data field, similar to TSF. However, Bencoding defines separate structures for lists (arrays) and dictionaries (hashes) and does not identify occurrences. It also imposes an order on dictionary strings. Binn [29], a more recent design, uses zero-copying and counts for structures, like TSF, but defines types using bit fields and varying length binary fields for data lengths and container lengths. Unlike TSF, it has a number of hard-coded data types, and three containers types, but it does make a provision for user types.

1.2 TSF Design Objectives

Now that TSF has been differentiated by what it is not, we follow the example set by CBOR [24] in RFC 7049 and list the design objectives of the Transaction Serial Format (TSF), in order of importance:

(1) The format must be efficiently deserialized.

(a) Deserialization should not require more than a few pages of C code.

(b) The format must not require parsing for deserialization.

(c) The format must not use data units smaller than a byte.

(d) The format must avoid forcing data into specific bit representations.

(e) Deserialization should not require a schema.

(2) The design must be general enough to encode popular data formats such as XML and JSON, as well as other common Internet formats.

(a) The format must support named and unnamed sequences, such as arrayed data.

(b) The format must support named and unnamed collections, such as hashes.

(c) Data structuring must support collections of sequences, and sequences of collections.

(d) TSF is not a streaming format; it does not support unspecified data lengths or occurrences.

(3) The internal (in memory) representation of a deserialized transaction is integral to efficiency of the design and must be easily created.

(a) Dynamic memory allocations should be minimized.

(b) Data should not need to be copied into the internal representation.

(4) The format should support user data typing to allow it to be adapted to specific user applications.

(5) The internal API must include a standard serialization.

(a) This is a common sense, ease of use objective.

(b) Contrast this with XML, which has no serialization API.

(6) The serial representation should be reasonably compact.

(a) Compactness is not a primary goal, but the equivalent JSON size is a compactness target.

(b) Redundancies such as end tags and extra delimiters should be avoided.

The primary goal of efficiency is a result of the simplicity of the TSF design. Figure 1 shows the complete lexical structure of TSF.

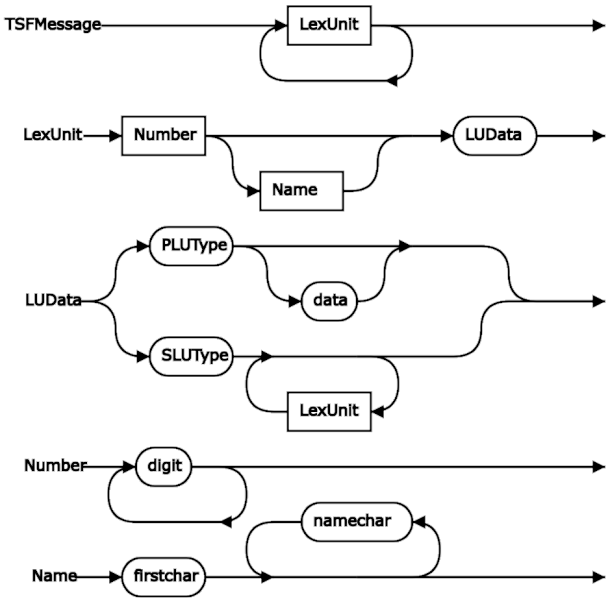

Figure 1. TSF Complete Lexical Structure

The following sections examine the design in detail and discuss the performance impact of TSF and its sister memory representation, TAM.

1.3 Summary

Section 1 (this chapter) introduces TSF and differentiates it from related works on serialization. It presents the goals that have guided the TSF design and summarizes the complete lexical structure in Figure 1.

Section 2 presents the detailed design of TSF and illustrates TSF message deserialization without parsing. The lexical processing is limited to recognizing a sequence of digit characters as a number, and scanning to the end of names. The lexical grammar that underlies the TSF design illustrates the simple one-character lookahead requirement. An informal proof shows the grammar produces TSF lexical units.

DOI: https://doi.org/10.30564/jcsr.v2i2.1620

Section 3 discusses the application of TSF to JSON. It generalizes the definition of a TSF name, and shows that it is possible to encode JSON data in equivalent TSF.

Section 4 presents TAM, and discusses the efficiency considerations that guide the design.

Section 5 applies TSF to XML. It shows that XML documents are a series of lexical units when represented in TSF, so that deserialization does not involve parsing.

Section 6 presents a comparison of the performance of deserialization of XML documents using the Expat C library implementations of the XML SAX parser, and the C/C++ implementation of TSF/TAM deserialization.

Section 7 summarizes the results and discusses the findings.

2. TSF Design

TSF provides a general serial transaction format. The TSF library deserializes a TSF transaction to its Transaction Array Model internal format with minimal processing by compact code. Specific design goals are:

(1) Minimal processing for message serialization and deserialization

(2) Simple in-memory representation of a deserialized TSF transaction

(3) Standard serialization and deserialization API's

(4) Reduction of redundant information, such as found in XML and JSON formats

For IoT devices, the benefits are:

(1) Reduced energy usage

(2) Smaller memory footprint

(3) Operations with slower, cheaper CPU's

2.1 Design

The elements of a TSF serialized message are lexical units (LU's), so called because recognition requires only simple lexical processing. There are two abstract lexical units. The first is a primitive lexical unit (PLU) that consists of a sequence of digit characters specifying the data *length*, a single type character that is not a digit, and *length* characters of data. TSF extracts a PLU from a message by lex'ing the length, recognizing the type, and extracting the corresponding data. The type character serves to distinguish various PLU types, as desired by the application using TSF. For example, type characters can distinguish between integers and floating point numbers.

A PLU can optionally have a name. In a named PLU, the name follows the initial length and ends with the type character. The name cannot begin with a digit and cannot contain any type character. In the case of XML-equivalent serialization, this is a simple restriction. XML tag name and attribute name come from a restricted character set. For complete generality, the restriction on name characters is removed with a convention described when we consider JSON, which places no restriction on the character composition of names.

The second type of TSF lexical unit is the structured lexical unit (SLU). An SLU consists of a sequence of digit characters specifying the contained unit *count*, a single type character, which is not a digit, and *count* subsequent LU's. An SLU is a container in the sense that the *count* identifies the number of immediately contained units. Knowing the *count* at the beginning of the unit allows the pre-allocation of needed storage. An SLU can also optionally have a name, with the name having the same location and restrictions as a PLU name. Each SLU-contained lexical unit may be any type, a named or unnamed PLU or SLU. In the case of XML equivalence, XML elements and lists of XML attributes are SLU's. For JSON equivalence, arrays and objects are SLU's. In a TSF message, named and unnamed lexical units occur in any combination.

Table 1 shows the syntax of the TSF lexical units. Syntax descriptions use the Augmented Backus Naur Form described in the IETF's RFC5234 [30].

Table 1. TSF Message Definitions

TSFMessage	=	1*LexicalUnits
LexicalUnits	=	PLU / SLU)
PLU	=	Length 0*1Name Type data
SLU	=	Count 0*1Name Type LexicalUnits
Length	=	Number
Count	=	Number
Number	=	1*digit
Type	=	a character that is not a digit or a name character
Name	=	does not start with a digit, or contain any type character

The actual characters used to indicate types can be chosen to reflect the particular application. Type characters and name characters are disjoint. As shown in Table 1, type and name characters have the following restrictions:

(1) types cannot be digits

(2) names cannot start with a digit

(3) type characters cannot be used in names

2.2 Lexical Simplicity

This section discusses the syntax that describes TSF lexical units and shows that it requires only the simplest kind of lexical processing, one-character lookahead.

The syntax descriptions shown in Table 1 exhibit the lexical simplicity of TSF. The lexical units of TSF can be recognized using only one lookahead symbol. Each lexical unit begins with a sequence of numeric characters. The

 DOI: https://doi.org/10.30564/jcsr.v2i2.1620

sequence is always terminated by a non-numeric character that is either the start of a name or a type character. If the type character implies a PLU, the numeric value is the number of characters that constitute the value. Thus, in the syntax description, this field, *Data*, is a terminal symbol. If the type character implies an SLU, the numeric value is the number of contained lexical units. This number also implies the storage needed for the associated TAM node.

Table 2 gives an alternative, right recursive, description of the TSF syntax. Instead of the ABNF zero occurrences syntax, an ε alternative explicitly indicates nonterminals that may be empty.

Table 2. TSF Right Recursive Syntax

TSFMessage	=	LexUnit LexList
LexList	=	ε/ LexUnit LexList
LexUnit	=	Number Name LUData
LUData	=	PLUType Data
LUData	=	SLUType LexList
Data	=	ε / data
Number	=	digit Digits
Digits	=	ε / digit Digits
Name	=	ε / firstchar Nchars
Nchars	=	ε / namechar Nchars

Figure 1 represent graphically the syntax of Table 2. In the diagram, a rectangle is a nonterminal symbol. A terminal symbol is a circle or elongated oval. The figure shows that the TSF syntax obeys the following two rules:

Rule 1: For any nonterminal, the set of first symbols for each of its alternatives is unique.

Rule 2: For any nullable nonterminal, such as *LexUnit* and *Name*, which have the empty string as an alternative, the set of its follow symbols is disjoint from the set of its first symbols.

Together, these two rules guarantee that any sequence of TSF lexical units can be unambiguously recognized by looking at the next terminal in the sequence (one symbol lookahead). Table 3 shows the first and follow sets.

Table 3. First and Follow Symbols for the TSF Syntax

Nonterminal	Nullable	First Symbols	Follow Symbols
TSFMessage	no	digit	
LexList	yes	digit	
LexUnit	no	digit	Digit
LUData	no	PLUType, SLU-Type	
Data	yes	data	
Number	no	digit	firstchar, PLUType, SLU-Type
Digits	yes	digit	firstchar, PLUType, SLU-Type
Name	yes	firstchar	PLUType, SLUType
Nchars	yes	namechar	PLUType, SLUType

2.3 Type Characters

TSF does not define specific type characters for *PLUtype* and *SLUtype*. These are user defined. The characters available for type characters are implied by the simple syntax. A type character cannot be a digit, and cannot be a character that may appear in a name. This is because lexically, the type character signifies the end of a number or a name. When names are defined as XML tag names or Javascript variable names, then all the remaining ASCII characters between 32 (0x20) and 127 (0x7f) are available for types. Specifically, these are the characters 32 to 47 (0x20-0x2f), 58 to 64 (0x3a-040), 91 to 96 (05b-0x60), and 123 to 126 (0x7b-0x7e). Although there is no lexical reason why 0 to 31 (0x00-0x1f) and 127 (0x7f) cannot be used, we avoid them to keep the TSF serializations text-editable. The characters in use by an application are initially set through API initialization.

In Section 3, since JSON already has a string representation for each of its primitive types, all primitive types are typed by a single "string" type using the single quote character (’). The character "[" is the JSON array type, and "{" is the JSON object type. The TSF serialization of JSON therefore requires three type characters. A different application could use more type characters to signify individual encodings of JSON primitive types.

Section 5 shows how TSF can serialize XML. In this TSF application, the type characters suggest XML meanings. The characters "[", "]", "!", "+", and "?" are all *PLUType* characters. The characters "<" and "=" are *SLUType* characters.

2.4 TSF Message Generation

In order to show that the TSF syntax of Table 2 is an accurate description of the TSF format, we show that the syntax generates TSF strings, with the following informal argument.

Beginning with a set consisting only of the goal symbol, *TSFMessage*, a new set is created containing all the strings generated by expanding *TSFMessage*. At each subsequent step, a new set of strings is generated from the previous set by replacement of the leftmost nonterminal of each string by each definition of the non-terminal. The intent of the process is to show that when a generated string contains only terminals, it is a correct TSF message.

(1) *TSFMessage* generates one or more *LexUnits*.

(2) *LexUnit* generates a *Number* followed optionally by a *Name*, followed by *LUData*. *Name* is optional since it may generate the empty string.

(3) *LUData* is either a primitive LU, if it starts with a *PLUType* (terminal) character or a structured LU if it starts with an *SLUType* (terminal) character.

DOI: https://doi.org/10.30564/jcsr.v2i2.1620

(4) A primitive LU is followed by *Data*, which may be *data* (a terminal), or empty.

(5) A structured LU is followed by zero or more *LexUnits*.

(6) A *Name* is a sequence of terminals.

(7) A *Number* is a sequence of terminals.

The informal analysis shows that a *TSFMessage* is a sequence of *LexUnits*, which is turn generate the terminal sequences of either primitive or structured lexical units. Therefore, the syntax generates only correct *TSFMessages*.

3. TSF JSON Equivalence

TSF's design is easily applied to JSON, the Javascript Object Notation. JSON is a character-based serial encoding of general computing data structures that requires parsing for deserialization. The web site "Introducing JSON" [18] describes JSON as

"... built on two structures:

A collection of name/value pairs. *In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.*

An ordered list of values. *In most languages, this is realized as an array, vector, list, or sequence."*

A JSON collection is equivalent to a TSF SLU containing a sequence of named PLU's and SLU's. A JSON array is a TSF SLU containing a sequence of unnamed PLU's and SLU's. TSF's internal data model, TAM, handles the JSON data structures. Several export convention allow TSF messages to be exported as either XML strings or JSON strings. We should note that the primary purpose of TSF is to transmit and receive messages with efficient serialization and deserialization. This description of the application of TSF to JSON underscores the generality of the TSF design.

3.1 Details of the Application of TSF to JSON

In order to apply TSF to JSON, we need:

(1) a PLU type for each of the JSON primitive types;

(2) a structured lexical unit type to identify arrays;

(3) a structured lexical unit type to identify collections;

(4) a convention for handling JSON names which contain type characters, or that begin with digits.

In all cases, we will use TSF type characters that suggest the JSON delimiters.

3.1.1 Primitive Types

Javascript's primitive types include integers, floating point numbers, strings, booleans, and a null type. However, JSON does not encode these types differently, but instead uses their string representations. With PLU types, it would be possible to encode each JSON primitive type differently in a TSF transaction. However, for the pedagogical purposes of this section, we keep the string encoding and indicate all primitive types with the single quote type character (').

3.1.2 Collections

A collection is the first of two JSON data structures, and is a sequence of named primitive and structured elements. This is exactly what a TSF SLU is, and we will use the left brace ({) as the SLU type character for a JSON collection.

3.1.3 Arrays

Arrays are second of the two data structures encoded by JSON. A separate SLU type character is needed for arrays, and we will use the left bracket ([) for arrays. The SLU count will be the number of elements in the array. As with all SLU's, an array can have a name, or it can be unnamed.

3.1.4 JSON Names Containing TSF Type Characters

TSF expects lexical unit names, if they are present, to exclude the characters that are used as type characters, (', {,[), so that the detection of a name can be accomplished by a short scan forward until a type character is encountered. In JSON, collection property names may contain any character. JSON handles this by delimiting all names using double-quotes. Double-quotes in a name are handled by escaping (preceding) a double-quote with a "\" character. Since the possibility of names containing the three LU type characters is remote, it is not necessary to incur the overhead of processing every name as if it possibly contained a type character. The double-quote character (") used by JSON is a good introduction character to signify that special processing should be done with a name. In a TSF serialization, a name that starts with a double-quote (") will be processed as if it contained '\' escape sequences. Any character following a '\' is accepted as a name character without further examination. An unescaped type character will terminate the scan normally. As an example, the four-character name {\[' will appear in a TSF serialization as "\{\\\[\'. If the double-quote character appears as a name character, it must be escaped.

There is one other subtlety regarding this convention. A non-digit character always terminates TSF *length* and *count* fields. In JSON, it is possible for a name to begin with a digit character, so the '"' name convention must also be used in this situation. Digits in a name are not escaped, but a name such as 64th must begin with the

 DOI: https://doi.org/10.30564/jcsr.v2i2.1620

double-quote, as "64th". This convention allows a name to be created from any sequence of characters, while keeping the name processing simple in most cases.

3.2 String Export

TSF serializations of JSON can be exported as either XML or JSON strings, by adopting certain conventions. This is more of an academic exercise than a practical one, but may be useful when a TSF message is exported from the realm of IoT to a different computing environment.

3.2.1 Exporting TSF as JSON Strings

Serializing a JSON string in TSF is in a sense a lossy transformation. Unlike XML, JSON ignores whitespace on deserialization, so parsing JSON loses whitespace formatting. This is the only possible difference between an input JSON string converted to a TSF representation and the output string converted from that representation back to JSON. If a JSON string has no ignorable whitespace, the conversion to TSF and back to JSON is lossless. However, TSF includes the possibility of preserving whitespace using unnamed text lexical units if the JSON-to-TSF converter recognizes and preserves whitespace.

3.2.2 Exporting TSF Serializations of JSON as XML Strings

If a TSF serialization only uses names that are valid XML names, it is always exportable as an XML string. With several conventions, it is possible to maintain exportability for JSON.

Unnamed Structured Lexical Units - An XML representation requires a name, so in those cases, other than arrays, where the JSON object is unnamed, a name can be generated from the position, nesting level and sequence, of the element.

Arrays - Array elements under the same parent have the same name, generated from the position of the parent. Optionally, the names can be unique by including the sequence number of the array element.

Arbitrary Property Names - Any name that does not obey the name rules for XML element names will have any invalid character converted to a five character sequence equivalent to the six character JSON UCS escape convention (\uhhhh), but with the leading '\' dropped. With this convention, a name will start with a letter, and contain only letters and digits.

Part II provides additional export conventions.

4. Transaction Array Model

The Transaction Array Model is the in-memory structure of a deserialized transaction. TAM is a simple, conceptually straightforward, representation of a TSF transaction. It features:

(1) minimization of the number of memory allocations needed to create the structure

(2) an array structure to minimize the data fields devoted to structure overhead

(3) use of the in-memory TSF transaction for data storage, also call zero-copying

TAM combines the lexical units that are the immediate content of an SLU into a single dynamically allocated node. This node has the structure of a small table. See Figure 2.

Row[1]	Type[2]	Name[3]	Length	Value[4]	Length
1	if SLU	name reference	Length	SLU reference	length
2	if PLU	name reference	length	PLU reference	length
...	PLU or SLU	...	...	...	...
n	...	...	...	...	...
[1] row numbers are not part of the structure					
[2] an 8-bit type character					
[3] null if no name, otherwise a direct reference to the TSF transaction memory					
[4] SLU: null if empty, or a TAM node reference					
[4] PLU: a direct reference to the TSF transaction memory					
Additional Fields in the Node					
a reference to the TSF transaction in memory					
a parent reference to the node containing the SLU that references this node					
a count of the number of rows allocated in this node					
a count of the number of rows used in this node					

Figure 2. A Transaction Array Model Node (TAMNode)

4.1 TAM Creation

TAM attempts to reduce the dynamic allocations needed to create the structure by taking advantage of the SLU occurrence counts embedded in a TSF string. Each SLU has an occurrence count for the number of directly contained LU's. TSF deserialization extracts these occurrence counts from the TSF string at the start of SLU processing. Each SLU count is the number of rows needed for the SLU's TAMNode. The full size of a TAMNode is thus computable as soon as the number of rows is known. The implications of this are different for each implementing language. For example, in C, all of the storage needed for a TAMNode is allocated with only a single dynamic memory request. In Java, where arrays must be allocated separately, more allocations are needed, but still, when compared to the number of allocations needed for an XML DOM representation, there is a significant reduction. This

DOI: https://doi.org/10.30564/jcsr.v2i2.1620

reduction in dynamic allocations in TAM translates to reduced memory and processing overhead.

Figure 3 shows the pseudo-structure of a TAMNode in C, with each of the attributes of the table declared in a separate array.

```
struct TAMNode
{
   char *tsfXact;
   struct TAMNode *parent;
   unsigned elemUsed;
   unsigned elemCount;
   char elemType[elemCount]
   char *elemName[elemCount];
   unsigned elemNameLen[elemCount];
   void *elemValue[elemCount];
   unsigned elemValueLen[elemCount];

};
```

Figure 3. A TAMNode

It is a pseudo-structure because in C, arrays, such as elemName, cannot be declared with a computable size. However, since the total number of lexical units (elemCount) is known before the structure is allocated, the actual amount of storage needed can be computed. The declaration in Figure 2 is informative. The actual structure uses double pointers to locate each of the variable length sections in the TAMNode so that they can be referenced within C code as simple arrays, even though they cannot be declared exactly as shown in the listing.

4.2 Value Storage

TAM also uses a compact approach to store names and data. A TSF transaction is read as a single string, contiguous in memory, and passed to a deserialize() method. Values and names are identified by their offset from the start of the string and their length. In this way, no extra storage or allocations are required. This is a language-neutral approach, and works for all languages such as Java, that do not use string terminators. For C, an alternative is available, if the transaction consists only of character data. The nature of the TSF lexical units allows each name and value to be 0-terminated. This is done on the fly as the TSF message is being processed. Names and values are then directly referenced as 0-terminated C strings. In either case, whether offsets and lengths or 0-terminated strings, names, and data values are all located in the original serialized TSF transaction and no additional allocations are required to store them. This is also true for generalized names (described in Part I Section 3) after escape sequence removal.

4.3 The TAM Root Node

The individual nodes of a TAM structure are linked through SLU references and parent references.

A TSF transaction always begins with an SLU that contains the entire transaction, somewhat like an XML root node. Figure 4 shows an example of this kind of node. There is a reference to the root SLU's name, and to the TAM node that contains all the LU's in the SLU.

When the transaction container does not have a name, as is the case in JSON transaction serializations, an optimization is possible. In this case, the only relevant data is the reference, the value field of row 1 in Figure 3. This reference can then become the reference to the new root node, and the original root node is optimized away.

Type	Name	Length	Value	Length
=	→name	len	→TAMNode	n/a
→tsfxact				
null (a parent reference)				
1 (number of rows allocated in this node)				
1 (number of rows used in this node)				

Figure 4. A TAMNode with a Single Unnamed SLU

An example of this is the transaction shown in Figure 5. The root SLU ('=') contains three LU's, two PLU's ('+,'!') and an SLU ('<'), and is unnamed. (The figure shows the embedded CR using the C escape convention '\n' which should be counted as a single character.)

3=9+ comment 31!doc [< !ELEMENT doc (#PCDATA)>\n]0doc<

Figure 5. A TSF Transaction With Three Top Level LU's

The root node, representing the SLU containing three LU's, would store a reference to the node of Figure 5. Since the only important field in the root node is the reference to the child node, the root node can be dropped. The child node reference becomes a reference to the new root node, Figure 6. If the top level SLU has a name, then this optimization cannot be used, because the name must be referenced in addition to the reference to the contained data.

Type	Name	Length	Value	Length
+	null	0	→"comment"	9
=	null	0	→"doc [< !ELEMENT doc (#PCDATA)>\n]"	31
<	→"doc"	3	null	-
→tsfxact				
null (a parent reference)				
3 (number of rows allocated in this node)				
3 (number of rows used in this node)				

Figure 6. The Effective Root Node With Three LU's

 DOI: https://doi.org/10.30564/jcsr.v2i2.1620

5. XML Equivalence: An application of TSF

This section presents an application of TSF to XML to demonstrate the generality of TSF by showing that XML documents can be completely represented in TSF. In addition to the demonstration of generality, we discuss XML equivalence to lay the foundation for performance comparisons. Several short sections discuss various XML issues such as DOCTYPE's and namespaces.

5.1 Relevant XML Definitions

The following definitions from the XML 1.0 recommendation [31] are relevant to the application of TSF to XML. The numbers in square brackets within the tables are the identifiers of the definitions in the XML recommendation.

5.1.1 XML Names

"A *Name* is a token beginning with a letter or one of a few punctuation characters, and continuing with letters, digits, hyphens, underscores, colons, or full stops, together known as name characters." [31]

Table 4. The Syntax of an XML name

[4]	NameChar	=	Letter / Digit / "." / "-" /"_" /":" / CombiningChar / Extender
[5]	Name	=	(Letter / "_" / ":") *NameChar

CombiningChar's and *Extender*'s are classes of Unicode characters that are not relevant to TSF. What is important for its design is that a *Name* (called an *XMLname* below) cannot contain the characters selected as type characters.

5.1.2 Element Names and Attribute Names

Table 5 shows XML definitions relevant to TSF attribute serialization.

Table 5. The Syntax of an XML Start Element Tag

[40]	Stag	=	"<" Name *(S Attribute) *S ">"
[41]	Attribute	=	Name "=" AttValue

The XML Recommendation's ABNF definition of *AttValue* uses character exclusion, which makes definition awkward, so we describe attribute values in English.

An *AttValue* can be any sequence of characters delimited by leading and trailing single quotes, or leading and trailing double quotes. Characters with XML syntactic meaning cannot be coded literally in an *AttValue*, but must be encoded using XML entity references. The relevant entity references for attribute values are *<*, *>*, *&*, *'*, and *"*, for the characters "<", ">", "&", single quote (0x27), and double quote (0x22), respectively. The *S* in the definition represents XML white space. TSF does not need entity references.

5.2 TSF Types for XML

TSF type characters identify the following XML types. The definitions all begin with lower case characters because they are terminal elements in the TSF syntax description of XML (Table 7). As terminals, they do not need any auxiliary processing, such as scanning.

xML-doctype-content - a DOCTYPE entity

xML-processing-instruction - a Processing Instruction

xML-comment - character data in an XML document following the opening four character "<!--" sequence and ending at the three character "-->" terminating sequence

xML-pcdata - parsed character data (XML PCDATA)

xML-cdata - character data (XML CDATA) (unexamined character data)

xML-attribute-content - the sequence of characters, which make up the value of an XML attribute; TSF do not require XML entities for attribute content.

5.2.1 Lexical Units

Table 6 shows the XML lexical units.

Table 6. The Syntax of an XML Document as a TSF Message

TSFXMLDoc	=	1*(SLU / PLU)
SLU	=	Element / Attrs
PLU	=	Doctype / ProcInst / Comment / Text / Cdata / Attr

XML elements and attribute lists are represented by Structured Lexical Units. Unstructured XML constructs are represented by Primitive Lexical Units, shown in Table 7 with their single character types. The number is a length indicating the number of value characters that follow the type character. The only PLU that has a Name is the *Attr*. The Name conforms to the XML definition.

Table 7 TSF PLU Syntax for XML

Number	=	1*digit
Doctype	=	Number "!" xML-doctype-content
ProcInst	=	Number "?" xML-processing-instruction
Comment	=	Number "+" xML-comment
Text	=	Number "[" xML-pcdata
Cdata	=	Number "]" xML-cdata
Attr	=	Number Name "[" xML-attr-content

DOI: https://doi.org/10.30564/jcsr.v2i2.1620

Note that *Text* and *Attr* use the same type characters, but there is no ambiguity because they occur in different containers.

XML elements (*Element*) and attribute lists (*Attr*s) are SLU's, shown in Table 8. With SLU's, the leading number is the count of contained lexical units. With the *Element* SLU, contained units include processing instructions, comments, parsed character data (PCDATA), character data (CDATA), and subsidiary (child) elements. The count is greater than or equal to zero. The occurrence of *Element* in the definition of *Content* provides the recursive definition for a nested XML data structure. With the *Attr*s SLU, the count is the number of attributes in the attribute list. If there are no attributes, there is no *Attr*s SLU, which is distinguished by its type code.

Table 8. TSFString Count Unit Syntax

Element	=	Number Name Attrs "<" Content
Attrs	=	*1(Number "=" 1*Attr)
Content	=	*(ProcInst / Comment / Text / Cdata / Element)

5.2.2 DOCTYPEs, Processing Instructions, Comments

A complete serialized format for XML transactions must handle DOCTYPE's, comments, and processing instructions that are outside the root element, as well as a single document root element. The complete *TSFXMLMsg*, Table 9, has an optional DOCTYPE followed by zero or more processing instructions and/or comments, one XML (root) element (*TSFXMLDoc*), and zero or more processing instructions and/or comments. The *TSFXMLMsg* is an unnamed SLU whose type character is '='. This overloading of the '=' character is not ambiguous since it is not contained in an *Element*. Most often, a *TSFXMLMsg* will be just the XML root element, composed of the lexical units of the *TSFXMLDoc* definition. In this case, the leading "1=" sequence can be heuristically implied and omitted. For additional information about the root node, see Section 4.3.

Table 9. TSFXMLMessage with Outside XML *Element*s

TSFXMLMsg	=	Number '=' *1(Doctype) *(Procinst / Comment) TSFXMLDoc *(ProcInst / Comment)

5.3 Issues

5.3.1 TSF Encoding

An XML document is normally introduced with the

<?xml version="1.0" encoding="..."?>

processing instruction specifying the XML version and the encoding of the following document. Until very recently, there was only one XML version, 1.0. The new XML 1.1 version handles unusual situations that do not affect the core of XML usage [32]

While the ability to exchange documents encoded with different encodings is useful, an IoT application will normally select a single encoding. The UTF-8 encoding is a superset of US-ASCII, UCS-2, and UCS-4, and is the encoding used for performance comparisons. The TSF/TAM design addresses limited capability devices, often found in sensor networks, and assumes that encoding, once decided, is not an issue.

The TSF design does not preclude the use of other encodings should an application have need for it. The application can design a leading PLU to convey encoding. The code accompanying this work comes in two versions: extended (8-bit) ASCII using 8-bit characters internally, and UTF-8 encoding using 32-bit (UCS-4) characters internally.

Eight-bit encoding is important in that it can support binary data in a TSF transaction. Since TSF does not parse data, it can have any 8-bit value, so TSF supports direct binary transmission.

5.3.2 Namespaces

XML namespaces are not given any special treatment by TSF. An XML name or attribute name may have a namespace prefix. The prefix-qualified name, when it occurs, is a normal *XMLname* in the format. Namespace URL's are represented in the normal way as attributes. With namespaces, the colon character becomes a name character and therefore cannot be a type character.

5.3.3 Character and Entity References

XML markup gives certain characters special meaning. The XML need for special sequences to provide for these characters as normal data characters has been mentioned above.

TSF does not parse data, so it has no need for these sequences. All data characters appear in a TSF string in their normal encoding. There is no additional escaping of special characters needed.

5.3.4 XML Document Reconstruction

The application of TSF to XML encompasses all XML documents, and supports a complete reconstruction of any TSF-encoded XML document. However, there are cer-

 DOI: https://doi.org/10.30564/jcsr.v2i2.1620

tain limitations to exact document reconstruction brought about by parsers and XML equivalences:

(1) The order of attributes is XML-parser dependent.

(2) The form of an empty XML element is optionally selectable.

(3) The preservation of whitespace outside the root element does not occur.

(4) The preservation of white space within an XML start element tag is parser-dependent.

(5) Line end sequences are optionally selectable.

5.4 The Overhead of TSF

Although the motivation for TSF is not compactness, a side effect of the format is a smaller transaction when compared to its XML equivalent.

PLU's each have one character indicating the type and a length field whose width is dependent upon the number of data characters in the unit. If *l* is the number of data characters, the width of the length field is [log 1], so, with the type character, the PLU overhead is 1+[log 1]

SLU's have the same kind of overhead. If c is the count of contained lexical units, the width of the count field is [log c], so, with the SLU type character, the overhead is 1+[log c].

A simple XML element whose name is *n* characters has an overhead of 5 + n. This reflects the 5 delimiter characters, 2 <, 2 >, and 1 /, plus an extra occurrence of the name in the end element tag.

An XML element with only PCDATA is equivalent to an SLU/ PLU combination.

Table 10 summarizes the overhead of various lengths of an SLU/PLU combination vs an XML element.

Table 10. TSF Overhead Compared to XML

		Data Length		
Unit	**Character Overhead**	**1-9**	**10-99**	**100-999**
SLU + PLU	3+[log datalength]	4	5	6
XML *Element* (length n name)	5 + n	5 + n	5 + n	5 + n

For a simple XML element whose name is *n* characters, the XML overhead is *n* + 5. The primary size difference between a TSF string and its equivalent XML is the missing redundant end tag name. The savings improves with the number of elements in a transaction. Section 6, Table 13 shows test file size comparisons between XML and TSF.

5.5 Examples

A few examples of TSF serialization are shown to illustrate the foregoing descriptions.

Table 11. TSF Examples

Form	**Serialization***
XML	<project/>
TSF	0project>
XML	<ns:personnel xmlns:ns="urn:foo"><ns:person id="-Boss"/><ns:person id="worker"/></ns:personnel>
TSF	3ns:personnel<1=7xmlns:ns[urn:foo1ns:person<1=4id[-Boss1ns:person<1=6id[worker>
XML	<?peri rset?><!--Introduction--><project>content</project><!--Epilog--><?peri sset?>
TSF	5=9?peri rset12+Introduction1project<7[content6+Epilog9?peri sset
	* line wrapping is not part of the serialization

5.6 Embedding TSF in XML

An XML Processing instruction can embed a TSF string in order to send it within an existing XML infrastructure. A program using the XML SAX API could then handle the TSF PI as a special case. Consider

<?tfx *TSFString*?>

As in all situations where control characters may be recognized as data, if XML is going to recognize the ending processing instruction two-character sequence '?>', it cannot appear in the TSFString. The sequence is never part of the TSF control structure, so the only potential conflict would be in application data.

6. TSF/TAM Performance

This section compares the performance of standard XML deserialization processing against TSF deserialization. The focus of the performance testing is on deserialization, as opposed to serialization, because, of the two operations, deserialization has a formal API. XML parsing has two standard deserialization API's, SAX and DOM. XML serialization depends upon the internal data model. Some libraries will provide serialization from DOM, but if one is using a SAX parser for performance, then one is also building a custom data structure from the parse, which implies that a serializer must also be custom. The TAM library provides a serializer, which could be compared to an XML DOM serializer, should one desire, but the following compares only deserialization performance.

6.1 Test Files

Table 12 shows the input test files and their IDs, used as reference in other tables. The ID's are assigned in file size order. The size in bytes and a brief description are included.

DOI: https://doi.org/10.30564/jcsr.v2i2.1620

Table 12. Test File Descriptions

ID	File Name	File Size	Description
F1	future001.xml	70358	Scenario file from the Mana Game Series
F2	bpmnxpdl_40a.xsd.xml	160946	XSD file for XPDL 2.0
F3	eric.map.osm.xml	218015	OpenStreetMap export from northern Wva
F4	cshl.map.osm.xml	298233	OSM export of a research laboratory
F5	sccc.map.osm.xml	404977	OSM export of a community college
F6	British-Royals.xhtml	482666	British Royalty Lineage from Alfred the Great
F7	csh_lirr_osm.xml	712661	OSM export of a train station
F8	exoplanet-catalog.xml	2147926	NASA Kepler Exoplanet Catalog
F9	LARGEbasicXML.xml	3420388	Military Strategy Game Unit Order of Battle

Table 13 compares the sizes of the XML test files and their TSF equivalents.

Table 13. Test File Size Comparisons

ID	XML Size	TSF Size	Reduction
F1	70358	54049	23.18%
F2	160946	142280	11.60%
F3	218015	206800	5.14%
F4	298233	284065	4.75%
F5	404977	386928	4.46%
F6	482666	477051	1.16%
F7	712661	677853	4.88%
F8	2147926	1456993	32.17%
F9	3420388	2797092	18.22%

Table 14 shows the XML characteristics of each file. Files F3, F4, F5, and F7 are similar and serve as a consistency check. Although differing slightly in size, the table shows that they have the same internal structure. The other files were selected because of their size and differing internal structures. Detailed explanations of the columns follow Table 14.

Table 14. Test File Characteristics

					Lex	Avg	Depth		Children	
ID	**Elems**	***Attr*s**	**DATA**	**Cmt**	**Units**	**Bytes**	**Avg**	**Max**	**Avg**	**Max**
F1	1936	6	2596	0	4538	11.9	3.5	7	2.2	251
F2	2565	3317	4011	29	9922	14.5	3.7	11	2.4	379
F3	2515	11021	2815	0	1631	12.8	1.6	3	1.3	2017
F4	3544	15360	3616	0	22520	12.8	1.6	3	1.3	2709
F5*	5135	20566	5294	0	30995	12.6	1.7	3	1.3	3523
F6	4589	391	7948	5	12934	36.9	10.2	15	2.0	3556
F7*	8630	36855	8915	0	54400	12.6	1.6	3	1.3	6385
F8	168728	420	66247	0	235395	6.2	6.0	7	1.4	4215
F9	73156	15989	146227	1	235373	11.9	4.6	6	3.0	1129
*** contains multibyte characters**										

Table 14 column explanations:

Elems - the number of individual XML elements in the document

*Attr*s - the total number of attributes on all the elements in the document

DATA - the total number of CDATA and PCDATA occurrences in the document

Cmt - the number of comments in the document

Lex Units - the total number of lexical units in the document, which should equal the sum of the previous four columns

Avg Bytes - (per lexical unit) the number of bytes in the document divided by the number of lexical units

Avg Depth - the average depth of the subtree below an XML element

Max Depth - the maximum depth of the document; the maximum n umber of elements encountered in the path from the root to the lowest leaf element

Avg Children - the average number of child elements for any given element (Max Children omitted from this calculation to avoid skewing the value)

Max Children - the maximum number of children parented by any element; in these documents, this is almost always the number of children of the root element

6.2 Performance of the TSF Implementation Compared to Libexpat

Libexpat [33] is a library, written in C, for parsing XML documents. It is a popular parser used in many industry-wide programs, including the open source Mozilla project, Perl's XML::Parser package, and Python's xml.parsers.expat module. It has undergone extensive development, testing, and release-to-release improvements. The release used for the following work is libexpat-2.2.6. The C compiler used to build TSF/TAM, libexpat, and the deserialization performance drivers on the MacBook Pro is:

Apple LLVM version 10.0.0 (clang-1000.10.44.4)

Target: x86_64-apple-darwin18.2.0

Thread model: posix

Processor: 2 GHz Intel Core i7

The performance tests were run on the same machine.

Note the following points when reading the information on comparative performance statistics between libexpat and TSF/TAM:

(1) Libexpat is a SAX parser. Libexpat XML file parsing uses minimal callback functions build a document tree, in order to correspond to the work done to build the Transaction Array Model when deserializing TSF.

(2) The SAX callbacks build a stripped-down DOM. In order to minimize memory allocation overhead, single

 DOI: https://doi.org/10.30564/jcsr.v2i2.1620

allocations are used for multiple strings. For example, to build an attribute element from name and a value, a single memory request is made and the null-terminated name and value are both copied into the allocation.

(3) Namespace processing in libexpat is disabled to correspond with the TSF design. In this situation, libexpat treats a namespace prefix-qualified tag name or attribute name as a single sequence of characters. *xmlns*-prefixed attribute names are not significant.

(4) CPU time is collected using the getrusage() C library function.

(5) All processing is done using in-memory input with no threading.

Table 15 shows Libexpat CPU times to deserialize each of the nine test files. There are five separate runs for each file and the mean and standard deviations for the runs are shown in the last two columns. Table 16 shows equivalent statistics for TSF deserialization operating on each of the test files over five runs. Note that this is an apples-to-apples comparison in that the TSF program is working with UTF-8 TSF files and supports a UCS-4 character set internally.

Table 17 shows the performance of an 8-bit character implementation of the TSF algorithm, using the seven input files that do not have multi-byte character input. As expected, the improvement is even better.

Table 15. Five Run Deserialization Performance Using a Libexpat C library (microseconds CPU time)

File	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stdev
F1	3051	2856	3213	3360	3126	3121.20	167.77
F2	7501	7046	7704	7786	7184	7444.20	287.69
F3	11413	11025	11053	11114	11287	11178.40	148.48
F4	15635	15704	15531	15117	16352	15667.80	398.15
F5	23402	22393	25996	21137	22454	23076.40	1627.62
F6	10001	10922	11212	11100	10938	10834.60	430.37
F7	42218	40058	44230	41468	39866	41568.00	1593.46
F8	114265	121418	125337	134613	128113	124749.20	6781.90
F9	158042	195949	180174	181541	185936	180328.40	12438.86

Table 16. Five Run Deserialization Performance Using a Wide Character C Implementation of the TSF Algorithm (microseconds CPU time)

File	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stdev
F1	662	585	579	551	550	585.40	40.85
F2	1678	1442	1398	1397	1402	1463.40	108.60
F3	2616	2053	2469	1887	1894	2183.80	302.43
F4	3134	2549	2836	2681	2592	2758.40	211.96
F5	4208	4166	3842	3663	4295	4034.80	240.97
F6	3367	2413	2670	3332	2482	2852.80	414.33
F7	7355	7097	6759	6549	6450	6842.00	339.00
F8	20513	20151	19287	21379	21208	20507.60	757.23
F9	35575	35133	29692	29965	37841	33641.20	3246.97

Table 17. Five Run Deserialization Performance Using a C 8-bit Implementation of the TSF Algorithm (microseconds CPU time)

File	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stdev
F1	409	402	402	411	401	405.00	4.15
F2	1285	1050	1050	1050	1050	1097.00	94.00
F3	1372	1468	1339	1302	1335	1363.20	56.90
F4	2081	1809	1814	2042	1813	1911.80	122.86
F6	1203	1102	1084	1083	1133	1121.00	44.82
F8	19242	23759	22204	19090	16728	20204.60	2485.44
F9	21972	21580	21138	20849	21530	21413.80	386.72

Table 18. Deserialization Performance Improvement Factor, TSF vs Libexpat

File ID	F1	F2	F3	F4	F5	F6	F7	F8	F9	Mean
TSF Improvement (UTF-8)	5.3	5.1	5.1	5.7	5.7	3.8	6.1	6.1	5.4	5.4
TSF Improvement (8-bit)	7.7	6.8	8.2	8.2	-	9.6	-	6.1	8.4	7.9

Table 18 summarizes the deserialization performance improvement provided by TSF (UTF-8). The improvement factor is the mean Expat CPU time divided by the mean TSF CPU time for each file. The overall mean improvement factor is 5.4, a reduction of the CPU time of more than 80%.

As an additional indication of the consistency of the results, the CPU times for both libexpat XML deserialization and TSF deserialization are highly correlated with the number of lexical units in each file, given in Table 14. For TSF, the correlation is 0.956. For libexpat, the correlation is 0.975.

The reduction in deserialization time for TSF by a factor of 5.4 in comparison to XML shows that TSF can be a significant energy reduction component of an IoT device that sends and receives structured data.

As an added bonus, the headers and source code for TSF/TAM total less than 600 lines. The source code for the libexpat XML parser is approximately 15,400 lines.

7. Conclusion

As the Internet of Things expands with limited capability devices, efficient transactions formats can help move data faster, and with less energy. Less energy means a longer field life for an IoT device that does not have an external power source. The Transaction Serialization Format provides such a format. It is general enough to support full XML document and JSON object serialization and deserialization for a small fraction of the memory and CPU cost, as demonstrated by performance analyses comparing

DOI: https://doi.org/10.30564/jcsr.v2i2.1620

a traditional XML library. The Transaction Array Model provides a simple internal memory structure for handling the lexical units of a TSF message. The TAM structures can be created and destroyed with fewer requests for dynamic memory than needed for the well-known XML Document Object Model, and at the same time are memory conservative.

The code supporting this work is available from https://github/dde/TSF.

References

[1] G. Pinto and F. Castor. Energy Efficiency: A New Concern for Application Software Developers. Communications of the ACM, 2017.

[2] Apache Group. Apache Avro. 2012. [Online]. Available at: https://avro.apache.org/docs/current/spec.html

[3] Apache Parquet. 2013. [Online]. Available at: https://parquet.apache.org/documentation/latest/

[4] MongoDB, Inc. BSON (Binary JSON). 2018. [Online]. Available at: http://bsonspec.org

[5] Java Language. Java Object Serialization. 1993. [Online]. Available at: https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html

[6] G. van Rossum. Python Pickle - PEP 3154. 2011. [Online]. Available at: https://www.python.org/dev/peps/pep-3154/

[7] L. Wall. PERL Modules DataDumper, FreezeThaw, Storable. 1991. [Online]. Available at: https://perldoc.perl.org/Storable.html

[8] The Object Management Group. CORBA. 2008. [Online]. Available at: https://www.omg.org/spec/CORBA/3.1/Interoperability/PDF

[9] The Java Language. Java Remote Method Invocation. 1993. [Online]. Available at: https://docs.oracle.com/en/java/javase/13/docs/api/java.rmi/module-summary.html

[10] H. Pennington et al.. The D-Bus Specification. 2003. [Online]. Available at: https://dbus.freedesktop.org/doc/dbus-specification.html

[11] Apache Group. Apache Thrift. 2007. [Online]. Available at: http://thrift.apache.org/static/files/thrift-20070401.pdf

[12] D. Winer. XML-RPC. 1998. [Online]. Available at: http://xmlrpc.scripting.com

[13] M. Gudgin et al., Eds. SOAP Version 1.2. 2007. [Online]. Available at: https://www.w3.org/TR/soap12/

[14] International Telecommunications Union. Specification of Abstract Syntax Notation One (ASN.1). ITU Standard (Blue Book), 1988. [Online]. Available at: https://www.itu.int/rec/T-REC-X.208/en

[15] K. McCloghrie, D. Perkins, J. Schoenwaelder, Eds.. RFC 2578 - Structure of Management Information Version 2 (SMIv2). 1999. [Online]. Available at: https://tools.ietf.org/html/rfc2578

[16] International Telecommunications Union. X.509 - IT OSI - Public-key and *Attr*ibute Certificate Frameworks. 2008. [Online]. Available at: https://www.itu.int/rec/T-REC-X.509

[17] T. Bray, J. Paoli, C. M. Sperberg-McQueen, Eds.. Extensible Markup Language (XML) 1.0. February 1998. [Online]. Available at: https://www.w3.org/TR/1998/REC-xml-19980210

[18] D. Crockford. Introducing JSON. 2005. [Online]. Available at: http://json.org/

[19] O. Ben-Kiki, C. Evans,d I. döt Net. YAML Ain't Markup Language - Version 1.2. 2001. [Online]. Available at: https://yaml.org/spec/1.2/spec.html

[20] R. Rivest. S-Expressions. 1997. [Online]. Available at: http://people.csail.mit.edu/rivest/Sexp.txt

[21] M. Cokus, S. Pericas-Geertsen, Eds.. XML Binary Characterization Properties, W3C Working Draft 05, 2004. [Online]. Available at: https://www.w3.org/TR/2004/WD-xbc-properties-20041005/

[22] J. Schneider et al., Eds.. Efficient XML Interchange (EXI) Format 1.0 - W3C Working Draft 16, 2007. [Online]. Available at: https://www.w3.org/TR/2007/WD-exi-20070716/

[23] J. Clark, J. Cowan, Eds.. MicroXML. October 2012. [Online]. Available at: https://dvcs.w3.org/hg/microxml/raw-file/tip/spec/microxml.html

[24] C. Bormann, P. Hoffman. Compact Binary Object Format. 2013. [Online]. Available at: https://tools.ietf.org/html/rfc7049

[25] Google. Protocol Buffers. 2008. [Online]. Available at: https://developers.google.com/protocol-buffers/docs/proto

[26] W. van Oortmerssen. Flatbuffers, 2014. [Online]. Available at: https://github.com/google/flatbuffers

[27] M. Eisler, Ed.. RFC 4506 - XDR - External Data Representation Standard. 2006, obsoletes RFC 1832.

 DOI: https://doi.org/10.30564/jcsr.v2i2.1620

[Online]. Available at: https://tools.ietf.org/html/rfc4506

[28] B. Cohen. Bencoding - Part of BitTorrent. 2008. [Online]. Available at: http://bittorrent.org/beps/bep_0003.html

[29] B. Ramos. Binn - Binary Data Serialization. 2015. [Online]. Available at: https://github.com/liteserver/binn/

[30] D. Crocker, Ed.. RFC 5234 - Augmented BNF for Syntax Specifications: ABNF. Internet Engineering Task Force Request for Comments, January 2008. [Online]. Available at: http://www.ietf.org/rfc/rfc5234.txt

[31] T. Bray et al., Eds.. Extensible Markup Language (XML) 1.0 (Fourth Edition) - W3C Recommendation 16 August 2006. W3C Recommendation, August 2006. [Online]. Available at: https://www.w3.org/TR/2006/REC-xml-20060816/

[32] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, J. Cowan, Eds.. Extensible Markup Language (XML) 1.1. August 2006. [Online]. Available at: https://www.w3.org/TR/2006/REC-xml11-20060816

[33] The Expat Development Team. LibExpat - Version 2.2.6. 2018. [Online]. Available at: https://libexpat.github.io

DOI: https://doi.org/10.30564/jcsr.v2i2.1620