A Scalable Framework for Representation and Exchange of Network Measurements

Jason Zurawski, Martin Swany
Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716
{*zurawski, swany*}*@cis.udel.edu*

Dan Gunter
Lawrence Berkeley National Laboratory
Berkeley, CA 94720
dkgunter@lbl.gov

Abstract

Grid and distributed computing environments are evolving rapidly and driving the development of system and network technologies. The design of applications has placed an increased emphasis upon adapting application behavior based on the performance of the network. In addition, network operators and network researchers are naturally interested in gathering and studying network performance information.

This work presents an extensible framework for the storage and exchange of performance measurements. Leveraging existing storage and exchange mechanisms, the proposed framework is capable of handling a wide variety of measurements while delivering performance comparable to that of less flexible, ad-hoc solutions.

1 Introduction

The process of collecting network measurements for use in distributed and Grid environments is desirable for enabling adaptive usage of resources, as well as for operational support and utilization information. Many tools exist to measure the various "characteristics" of the network, such as *bandwidth*, *delay*, and *loss* [15]. Statistical information derived from these measurements is needed for predicting future performance, and for the tuning of networked applications. However, without a consistent and readily available set of names for, and representations of, this diverse pool of information, analysis spanning multiple organizations and network measurement infrastructures are difficult to perform as well as validate.

In this paper, we describe and investigate an extensible system for storing and processing performance information in distributed environments, such as the Grid. We limit the scope of our description and results to the system's primary goal of storing and processing network metrics, but note that the design is applicable to other types of performance information. The internal design of such a system consists of two major parts: syntactic and semantic conventions dealing in the representations of various network measurements and a generalized access mechanism to the underlying information. As the framework itself remains language neutral, this work explores two implementation strategies built around the data representation.

As we are focused on programmatic access to measurements, storage and exchange formats play a crucial role in the overall design of the system. To scale, the system must be able to take advantage of inheret redundancies in the data access patterns. The most basic of these, for time-varying monitoring data, is the separation of infrequently changing metadata from frequent, time-sensitive, data. Many proposed exchange formats, such as the GLUE [9] schema XML representation and earlier work by the NM-WG, leave this separation up to the implementation. In contrast, the data exchange format presented here explicitly separates metadata from data.

This explicit separation has several important benefits. In stable storage, it lends itself to a more normalized layout for the measurement. On the wire and in a Web Services [29] context, the basis for an "include by reference" mechanism is formed, allowing implementations to eliminate redundant information in a way that is independent of the specific data representation. This in turn simplifies the delivery framework, which need only worry about efficient encodings.

The second contribution of this work is a consistent approach to data exchange format extension and evolution, in the context of XML. The basic idea is not new and in the area of network measurements dates back at least to the design of SNMP [23]. But we update this approach to use Web Services-friendly identifiers (URI's instead of OID's) and arrange our schemas so that the "required" elements are minimized. This allows new measurements to easily and independently extend the basic framework.

The formats employed by this work were developed as part of continuous work within the Global Grid Forum (GGF) Network Measurements Working Group (NM-WG) [16] and are currently used in several other projects [1, 10, 17, 22, 25]. The authors composed a subgroup of NM-WG that designed this schema. The implementations presented here are not the result of input from the NM-WG.

This paper will proceed as follows: Section 2 presents the obstacles and design considerations to this work. We lay out the major architectural considerations in Section 3, followed by details specific to our implementation in Section 4. The results gathered from a series of experiments are featured in Section 5. Section 6 relates some of the previous contributions in this field of research; we present concluding remarks in Sections 7.

2 Problem Statement

There are potentially conflicting design goals that motivate this work, such as the tension between interoperability and flexibility. Agreeing on standard mechanisms for sharing data in a large and diverse group like the GGF [8] has made it clear that a single interface and storage format is difficult to define, as there are many different environments in which performance information is gathered, used, and encoded. Of course, any solution which is so rigid as to preclude the inevitable advances in this area will not be successful. Challenges aside, the goals of our measurement and monitoring framework must facilitate:

- Normalized data encoding in canonical formats
- Extensibility to new data sources
- Flexible re-use of basic components
- Use of existing solutions and technologies
- Language/Implementation independence

One key facet of our problem is the apparent trade-off between extensibility and efficiency for Grid performance monitoring systems. On the one hand, the Grid community has adopted World Wide Web Consortium (W3C) [28] standards, such as eXtensible Markup Language (XML) [31] and Simple Object Access Protocol (SOAP) [24] to enable portability and interoperability. On the other hand, we know that the performance of the information system is important in that overhead incurred there affects the performance of the entire system. Additionally, there are the storage requirements of the data. Storage of a large number of encoded data elements, all of which demonstrate a similar pattern yet contain different information, is inefficient. [1] We address these two conflicting goals in turn.

2.1 Measurement Representation

The basic goal of the storage and exchange formats portion of the framework we present here is to allow the separation of rapidly changing information, henceforth the "data", from relatively constant information, or the"metadata". For example, a network *traceroute* would have as data the IP address and time of each network probe, and would have as metadata the source and destination host of the entire probe along with the tool used, its parameters, etc. This economy leads to efficiency. Metadata can then be stored, searched, and transmitted separately from the more dynamic data. Identifiers for explicitly linking the metadata and data sections, even when they do not appear in the same physical location, are naturally considered in this framework.

[1] Clearly, this type of storage is quite compressible, but that can cause problems for searching, etc.

A secondary goal is re-usability within the broader scope of grid information exchange. Many information exchange schemas, including earlier versions of the GGF NM-WG schemas, had separate request and response sub-schemas. The approach presented here separates the semantics of the exchange pattern from that of the data representation. That is, a common representation of data and metadata is used for both requests and responses, simplifying the schema considerably and allowing for subsequent re-use of base definitions. This becomes even more desirable if you consider communicating measurements in a notification framework such as WS-Notification [30].

2.1.1 Measurement Encoding

XML provides the capability to produce self-describing documents. This has many advantages, but efficiency is not one of them. In the words of the XML 1.0 specification, "Terseness in XML markup is of minimal importance" [31]. This inefficiency makes both serialization and deserialization much slower than more machine-friendly formats. However, the gains in interoperability from text-based self-describing formats are also important for large distributed systems, as evidenced by the explosion of XML representations and toolchains in this area. We attempt to strike a happy medium by minimizing the redundant elements in the XML representation, and, when even that won't do, including support for specifying out-of-band mechanisms (e.g., binary formats) for transmitting the bulk of the data.

This approach to encoding can also be viewed as normalization. By storing the data entries in a normalized fashion and referring to external metadata as appropriate, we can address all our stated problems. By providing for simple recurring event storage in a minimal format, we can support high-performance data transfers and the automatic assembly of complex, self-identifying XML structures for simple applications and for human consumption. We will present below performance results that show that the schemas described here impose minimal overhead in comparison to raw SQL operations.

3 Schemas

The framework we present here is comprised of two major parts: the XML schema definition for measurement instances, and the software designed to store and deliver these instances on demand. We present a basic overview of the general schema with the understanding that specialized schemas may be developed from this initial pattern to fit the many different tools and characteristics of network measurement. This discussion will be followed by an overview of the prototype we have designed to manage interface utilization data.

3.1 XML Schema Language

The standard serialization format for the NM-WG data is XML. Therefore the primary representation of an NM-WG schema is an XML schema language, in this case the OASIS standard RELAX-NG [20]. We chose this language for its readability and elegance, keeping in mind that it may need translation to XML Schema [32], the most commonly used schema language. Several tools exist to perform this translation where appropriate.

A primary reason for using RELAX-NG was its intuitive and readable "compact" syntax.[2] Because some readers may not be familiar with this syntax, a brief summary follows. Allowed elements and attributes are prefixed with the keyword 'element' and 'attribute', with their datatype enclosed in {curly braces}. Maximum and minimum number of repetitions given with familiar regular-expression symbols of '?' for zero or 1, '*' for zero or more, and '+' for one or more. Elements and attributes can be joined by either a ',' indicating sequence, a '&' indicating an unordered group, or a '|' indicating a choice. Arbitrary groups of the above patterns can be assigned a name using the '=' operator.

3.1.1 NM-WG Base Schema

The major components of base schema are illustrated in Figure 1. In this figure, the major sections, data and metadata, are shown side-by-side with the subsections listed vertically within each section.

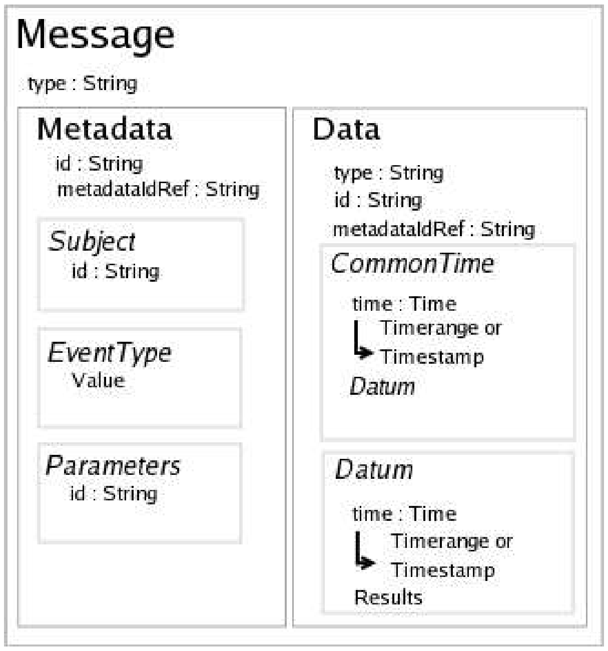

Figure 1. NM-WG Base Schema

[2]A Similar syntax summary of XML Schema would almost certainly occupy several pages.

The schema for the top-level message envelope is shown below.[3] The message envelope may contain multiple metadata and data sections. The message "type" allows distinguishing between storage and query, for example, when the underlying communication system may not provide such information.

```
namespace nmwg =
     "http://ggf.org/ns/nmwg/2.0/"

element nmwg:message {
     attribute type { xsd:string } &
     ( Metadata | Data )+
}
```

The schema for the Metadata element is shown below. Every metadata element must contain an "id" and may contain an optional "metadataIdRef" (formerly "metdataId"), which refers to another metadata section. This is to allow the Metadata elements to be "linked" for further reduction in storage overhead.

The metadata section is subdivided into three parts, only the first of which is required:

- Subject – The physical or logical entity being described. For example, a host pair or router address. Like the subject of the sentence: *Host A to Host B* measured ICMP latency is 100ms.
- EventType – The canonical name of the aspect of the subject being measured, or the actual event (i.e. "characteristic") being sought. Like the object of the sentence: Host A to Host B *measured ICMP latency* is 100ms.
- Parameters – The way in which the description is being gathered or performed. For example, command-line arguments to *traceroute* or whether the round-trip delay packet used ICMP or UDP. Like the descriptive clause of the sentence: *When you use* 100 *byte packets,* Host A to Host B ICMP latency is 100ms.

```
namespace nmwg = "http://ggf.org/ns/nmwg/2.0/"

Metadata =
  element nmwg:metadata {
    attribute id { xsd:string } &
    attribute metadataIdRef { xsd:string }? &
    Subject &
    EventType? &
    Parameters?
  }

Subject =
  element nmwg:subject {
    attribute id { xsd:string }
  }

EventType =
  element nmwg:eventType {
    text?
  }

Parameters =
  element nmwg:parameters {
    attribute id { xsd:string }
  }
```

[3]In all the schemas presented inline, some small details have been left out or modified to enhance readability. Full schemas are available at [26]

The metadata schema would validate the XML instance below. As in this example, the actual value of something with an identifier can be omitted for efficiency where it is provided by other context.

```
<nmwg:metadata id="1">
  <subject id="2"/>
  <nmwg:eventType>latency.oneway</nmwg:eventType>
</nmwg:metadata>
```

The schema for the data section is shown below.

```
namespace nmwg = "http://ggf.org/ns/nmwg/2.0/"

Data =
  element nmwg:data {
    element id { xsd:string } &
    element metadataIdRef { xsd:string } &
    (
      CommonTime? &
      Datum*
    )
  }
CommonTime =
  element nmwg:commonTime {
    Time &
    Datum*
  }
Datum =
   Time
}
```

Time is fundamental to network measurements, and is the only required part of each datum. The 'CommonTime' section allows the common case of factoring out a set of data that is associated with a single time range or timestamp. Note that by extending the EventType of the name into the namespace, effectively creating a unique name for each type of event, the timestamp may be all that is necessary.

Time-related elements reside in a sub-namespace from the base. This separation makes the time definition more portable, for re-use in extension namespaces. It also adds flexibility, allowing the time representation to change independently of the base namespace. The schema for the time namespace is shown below.

```
namespace nmtm =
  "http://ggf.org/ns/nmwg/time/2.0/"

Time =
  element nmtm:time {
    attribute type { xsd:string } &
    (
      TimeStamp |
      (
        StartTime &
        (
          EndTime |
          attribute duration { xsd:string }
        )
      )
    )
  }

StartTime =
  element nmtm:start {
    attribute type { token } &
    attribute inclusive { token }? &
    TimeStamp
  }

EndTime =
  element nmtm:end {
    attribute type { token } &
    attribute inclusive { token }? &
    TimeStamp
  }

TimeStamp =
  attribute value { xsd:string } |
  element nmtm:value { xsd:string }
```

3.1.2 Schema Extension

The abstract schema will be extended to represent the data returned from actual measurements. We use XML *namespaces* to allow independent extensions of the schema to co-exist without central coordination or "vetting". A namespace is a specific Uniform Resource Identifier (*URI*) that is similar to a Uniform Resource Location (*URL*) resembling the well known format `http://www.domain.org`.

The basic approach is to replace the base schema's elements with elements of the same name, but in the namespace of a specific organization. For example, if members of a school's computer science department create a new schema, it should be referred to as a subset of a domain they have access to, i.e. `http://cis.udel.edu/ns/new/tool/1.0/`.

In addition, the namespace construct can represent different versions of the same tool, or different schema versions through the implicit naming scheme. This feature fosters ease of transition between extension namespaces in the face of changing tools and measurements.

Building on the base schema section above, we present a subset of the interface utilization schema used in our implementation. This schema is capable of describing the specifics of a network interface, although for testing purposes the schema remains relatively simple in relation to "real-world" needs.

```
namespace utilization =
  "http://ggf.org/ns/nmwg/characteristic/util/1.0/"
namespace nmwgt =
  "http://ggf.org/ns/nmwg/topology/2.0/"

include "nmbase.rnc" {
   Subject = UtilizationSubject
}

UtilizationSubject =
  element utilization:subject {
    attribute id { xsd:string } &
    Interface?
  }

Interface =
  element nmwgt:interface {
    element nmwgt:ipAddress {  {
      xsd:string &
         attribute type { xsd:string }
    }? &
    element nmwgt:hostName { xsd:string }? &
    element nmwgt:ifIndex { xsd:string }? &
    element nmwgt:type { xsd:string }? &
    element nmwgt:direction { xsd:string }?
  }
```

The schema would validate for an XML instance such as this:

```
<nmwg:metadata id="1">
  <utilization:subject id="2">
    <nmwgt:interface>
```

```
      <nmwgt:ipAddress type="v4">
        128.4.133.163
      </nmwgt:ipAddress>
      <nmwgt:hostName>
        moonshine.pc.cis.udel.edu
      </nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
    </nmwgt:interface>
  </utilization:subject>
  <nmwg:eventType>ifInOctets</nmwg:eventType>
</nmwg:metadata>
```

This extension schema, and subsequent example instance documents, make up the first part of our framework. Constructing tooling to accept, parse, and extract meaning from these instances is addressed in the next section.

3.2 System Design

The storage and exchange format defined previously forms the basis for the service we present. Input and output to the service consists of messages containing varying amounts of information. With this requirement well defined, we must accomplish the goals of physically sending each message across a transport medium, parsing useful information from the transient messages, and performing the "task" assigned to each message. Our service chooses to implement the tasks of "storage" (accepting data to store internally) and "request" (returning data for a known pattern). The following sections lay out general solutions to the issues at hand. Specifics to our architecture will be featured in Section 4.

3.2.1 Message Transport

Interoperability with existing technologies is always a consideration when designing new software for the Grid. Creating a proprietary encapsulation and transmission protocol would not benefit the community. An obvious choice for transmission of application layer messages is SOAP. Care was taken when designing our SOAP bindings, specifically to the message encoding format. One must proceed cautiously, as there are only four acceptable ways to accomplish the same goal. These four choices arise from two independent choices: "RPC" or "document" structure, and "literal" or "encoded" XML.

To construct a message it is necessary to choose a structure and an encoding. A description of each is well beyond the scope of this paper; for our purposes, we consider the two most common methods: *RPC-Encoded* and *Document-Literal*.

As suggested by the name *RPC-Encoded*, the abbreviation "RPC" indicates a remote procedure call, and the word "encoded" refers to the use of a specialized set of XML types designed to represent programming language constructs such as arrays and directed graphs – structures that are not fully representable in XML Schema languages. For this and other reasons *RPC-Encoded* is on its way out due to interoperability issues in many cases.

Document-Literal places all the structure for the message (the whole "document") in the same place (the schema definition) as is; this method is thought to be a "cleaner" and simpler approach. For interoperability and simplicity *Document-Literal* is the best approach to start with. However, many toolkits, particularly ones that focus on simple client-side APIs, support only the older *RPC-Encoded* style.

3.2.2 XML Parsing

SOAP messages themselves are constructed with XML, and when using the *Document-Literal* format, contain our verbatim message. A natural thought progression leads to the suggestion of bypassing all "official" SOAP parsing software on the client and server ends in favor of creating a custom parser to handle our message format, as well as the additional SOAP tags. This saves the step of needing two rounds of parsing on the same data.

Our experiences have led us to settle upon classic parsing implementations of well known parsing APIs; the Document Object Model (DOM) [33] implemented in Perl [18] and a variant of the SAX [21] API, cElementtree [7], in Python [19]. Perl's implementation of DOM, for example, features a "document oriented" way of dealing with an XML instance. Efficiency is not a strong suit of DOM, because the entire document must be loaded into main memory upon parsing. As the document size grows, performance will certainly suffer. In sharp contrast, the Python cElementtree XML parsing library features an iterative parser that, like SAX, can process XML documents in a streaming fashion, retaining in memory only the most recently examined parts of the tree [14]. Streaming approaches are, of course, much more efficient for large XML documents

3.2.3 Information Storage and Retrieval

Back-end storage is an important consideration in any system. Speed of insertion and searching can easily become a bottleneck in high demand systems. Three basic options exist for storage: Relational database, XML database, flat file storage.

Relational databases features a well known and easily programmable interface, reasonable performance, and wide acceptance. Finding API bindings for the major database vendors is a simple task, and all major operating systems support some form of RDBMS. XML databases are an emerging technology that support the insertion and ability to index based on XML elements. At the present time there are few offerings from this realm and bindings exist only for a handful of languages. Flat files are of course an easy and accessible solution, but will require programmatic intervention to monitor and keep track of the location of specific information.

Future incarnations of our framework will no doubt explore these new technologies directly, and may utilize a hybrid approach as demonstrated with our XML evaluation. Utilizing a single RDBMS makes sense from both an interoperability and performance standpoint at this current point in time.

4 Implementation

Driven by the analysis in the previous section we have implemented a client/server architecture capable of storing and delivering XML messages conforming to the NM-WG schemas. Figure 2 describes the conceptual design of our measurement framework. We have implemented both the client and server portions and have utilized the service to exchange interface utilization data.

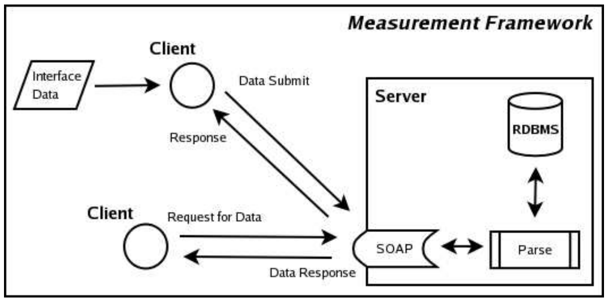

Figure 2. Framework Overview

4.1 Server

The server was implemented in the Perl programming language. Perl features rich APIs for XML parsing, SOAP operations, and HTTP server capabilities. As stated in the previous section, support for *Document-Literal* is limited in most implementations, and Perl is no exception. A customized HTTP server was implemented for the receipt and transmission of SOAP messages. SOAP libraries were used for the sole purpose of creating envelopes to send data between components.

XML processing on the server side consists of extracting the message from the SOAP envelope, and using DOM to parse the metadata blocks and related data blocks (in the case the message is meant to store information). Request messages are understood to contain no reference to data, so metadata alone is extracted. Information is gathered and formed into SQL statements ("insert" statements when we are storing data, and "select" statements for queries). When requesting information the database will return relevant results which are encoded into XML before being sent back to client applications.

4.1.1 Database

The MySQL database management system was utilized in this work. This database was chosen for its efficiency, size, and API interaction. A single table, wherein each row storing both metadata and data was constructed. Although this method consumes more storage space, we avoid the need to join multiple tables in the case of a query. Arbitrarily large XML dataset requirements may force future versions to implement multiple tables within the database.

The consideration of other storage technologies, such as XML and Object Oriented databases, are beyond the scope of this project. Future work will no doubt focus on the nuances of storage within the system in hopes of achieving better overall system performance.

4.2 Client

Client applications must have the capability to create XML messages in the NM-WG format, wrap these messages in SOAP envelopes, and contact a known server. The response from the server will also be in XML format; parsing software must be employed to extract meaning. Two clients have been constructed thus far; one implemented in Perl, the other in Python. Each client is interoperable with the Perl server.

4.2.1 Perl Implementation

As described in the Perl server, the Perl client uses the same basic SOAP and parsing operations. The client does not need to implement an entire HTTP server, but must send its XML message through a socket to the known address of the server. A response is also accepted through the socket. After receipt it is parsed for meaning and can be displayed, or the output may be funneled to a variety of other applications. For example, [25] uses interface utilization information to construct network "weathermaps" (graphical representation of network utilization) as well as utilization graphs over a time range.

4.2.2 Python Implementation

For efficiency, the Python implementation uses a mostly hand-rolled Web Services stack that is a combination of Frederick Lundh's cElementtree XML parsing library [7], and the standard Python HTTP library. The implementation is simplified in several ways, but as a result the entire web service stack was implemented in only a few hundred lines of code. Even though Python is a compact language, typical SOAP libraries still run in the thousands of lines. With elementtree, serialization and parsing are both incremental, and therefore memory usage is minimal.

5 Experimental Results

To test the performance of this framework we present tests of the Perl and Python client applications requesting datasets of various sizes from the server. A control test has also been designed to request the same data sets but through simple SQL requests directly to the database server (thus lacking all XML processing steps).

We performed this experiment over a wide-area network connection between Lawrence Berkeley National Laboratory (LBNL) and the University of Delaware (UD). The latency on this path was approximately 75 milliseconds, and the (TCP) bandwidth as measured by *iperf* [11] was about

30Mbits/second. The client host, at LBNL, was a 2GHz single-processor AMD Athlon XP 2400+; the server host, at UD contains dual 2.40GHz Intel Xeon processors. Both systems are running Debian [6] linux with a 2.6 kernel. The server is running a single database for these tests (MySQL version 12.22) as we aim to show the relative performance in a controlled environment. Future considerations will be given to various storage techniques besides that of typical relational databases.

Each implementation performed the same 5 different-sized queries, returning result sizes from 1 to 10,000, with three variations on each asking for information from 1, 2, or 3 router interfaces. Thus, the total amount of data returned ranged from 1 to 30,000 items. Each of these (15) different queries was repeated 5 times.

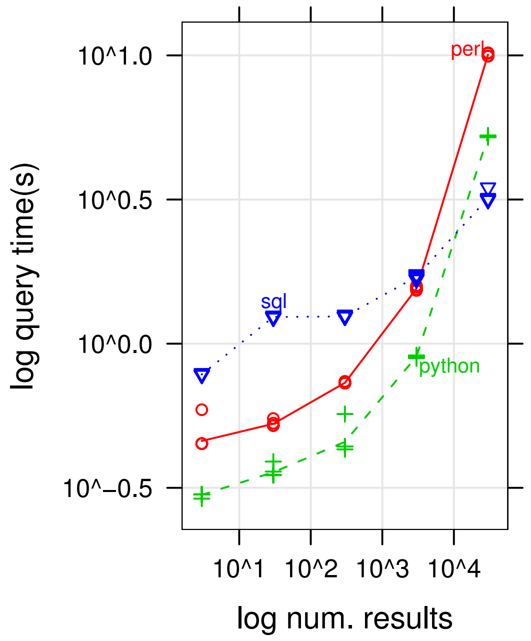

Figure 3. Query Size vs. Query Execution Time

Figure 3 exhibits the query performance of all three clients, for the three-interface variation of the query only. Analysis revealed an almost identical pattern of results for one and two interfaces. In each case, the SQL client was surprisingly slower than both the Perl and Python implementation for smaller numbers of results, becoming comparable for result sets in the hundreds and then becoming the fastest for thousands of returned items. In other words, the SQL implementation is shifted upwards but with a flatter slope, which indicates additional per-query setup time. This pattern is more pronounced with 2 and 3 interface queries, partially because the SQL implementation performed separate queries for each interface, whereas the NM-WG schemas naturally carried multiple metadata sections (with a query of an interface in each) in a single envelope. The smooth lines are calculated with Friedman's "super-smoother" algorithm [27]

The percent overhead, pictured in Figure 4, illustrates the "crossover" point between SQL and the XML implementations more dramatically, partially because it is a log-normal scale whereas the previous graph used a log-log scale (to help show the linear growth of query time vs. results). Previous graphs demonstrated that there was very little variation within a set of repeated measurements and that the pattern of query times is very similar across the number of interfaces being queried. Therefore, we can compare the medians of the times for a given total number of results (number of results * number of interfaces) to derive the percent overhead for Python and Perl relative to SQL. Again we use the "super-smoother" to help reveal the pattern of the results.

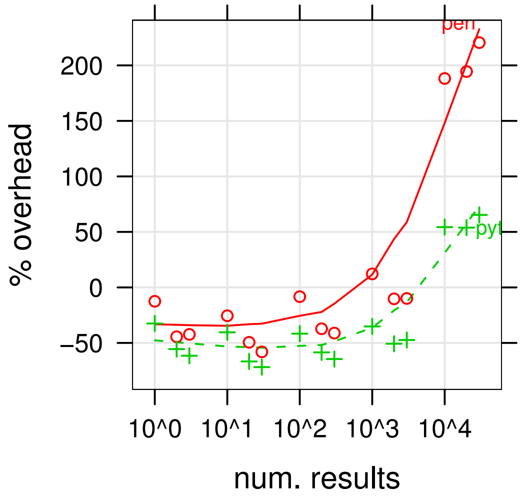

Figure 4. Overhead of Perl and Python relative to direct SQL interaction

5.1 Analysis

The unusually poor times demonstrated by the SQL client can of course be avoided. Experimental considerations mandated "equal-footing" for the definition of database queries. The SQL client essentially performs the same queries that a request encoded with a metadata block would invoke, and there is no ability to streamline many requests in a single statement executed across the WAN.

The performance gap between the Perl and Python implementations can be attributed to parsing technology. The DOM implementation of Perl requires significant memory, more so than parsers capable of reading directly from a stream, such as cElementtree in Python. Keeping a structure containing large result messages in limited memory space will inevitably perform more poorly than processing the message as a stream.

6 Related Work

In [2], the notion of a scalable system that enables the sharing of measurements was explored. The focal point of this work revolved around the sharing of entire experiments; we present our system in terms of individual measurements independent of specific experimental work. Overall our work shares the common idea of striving to make diverse measurements available, although our approach through the utilization of the NM-WG schemas offers a uniform storage and exchange mechanism; this simplifies the client and server interaction as well as database requirements. As this work did not produce a prototype, performance comparisons are not possible.

The IETF IPPM Working Group [12] aims to define metrics that will be used to describe various internet data delivery services and techniques. Recent work has been done within the group to develop a registry [13]. Similarly, we plan to construct a repository for the registration and storage of schema definitions that build upon the NM-WG base schemas.

The CAIDA [3] effort is focused on the collection, analysis, and dissemination of internet measurements. CAIDA established an "Internet Tools Taxonomy" [5] to aid in the definition and categorization of measurement tools that could be quite useful as a basis of the namespaces used in this system. Our future plans include incorporation of this taxonomy into the GGF namespace. Additionally CAIDA has begun to archive and share network measurements and is developing a schema for that effort [4].

The work we present here is unique in our early adoption of NM-WG schemas as well as the emphasis we place on system performance. These two characteristics lead to a scalable system in two ways: the system can scale to managing many data sets and handling numerous requests, additionally it is able to scale to the development of new and diverse measurements as defined by the NM-WG.

It is true that the approach of splitting data and metadata into tractable units is not new due to the breadth of work done by others in this field. We contend that in the world of web services, and more specifically the Grid, this technique is often not pursued. This attempt has shown that not only can it be done properly, the relative efficiency of the approach will guarantee performance and scalability.

7 Conclusion

We have presented a framework capable of storing and delivering network measurements.This framework separates metadata from data, providing a normalized and efficient means for transmitting and storing many measurements. We have shown how this approach promotes efficient and interoperable systems for exchanging performance information in Grid environments.

Our approach allows full use of the Web Services separation between schematic representation of the data and the transport protocols used to send it between parties. This allows the efficiency of data/metadata separation to be augmented, where desired, by efficient and appropriate wire formats.

Our framework features good performance compared to that of common ad-hoc solutions, despite needing to send and parse XML documents of various sizes and levels of complexity.

8 Acknowledgments

The authors would like to extend thanks to the members of the Global Grid Forum's Network Measurement Working Group. Without their encouragement and insight this work would not have been possible.

References

[1] Network Performance Advisor. http://dast.nlanr.net/Projects/Advisor/.

[2] M. Allman, E. Blanron, and W. Eddy. A scalable system for sharing Internet measurement. In *Passive and Active Measurement (PAM)*, March 2002.

[3] Cooperative Association for Internet Data Analysis. http://www.caida.org/.

[4] ISMA Data Catalog 2004 Workshop. http://www.caida.org/outreach/isma/0406/index.xml.

[5] CAIDA internet tools taxonomy. http://www.caida.org/tools/taxonomy/.

[6] Debian Linux. http://www.debian.org/.

[7] The cElementTree Module. http://effbot.org/zone/celementtree.htm.

[8] Global Grid Forum. http://www.ggf.org.

[9] GLUE Schema. http://www.globus.org/toolkit/mds/glueschemalink.html.

[10] INCA Test Harness and Reporting Framework. http://inca.sdsc.edu/.

[11] Iperf. http://dast.nlanr.net/Projects/Iperf/.

[12] IETF - IP Performance Metrics (IPPM). http://www.advanced.org/IPPM/.

[13] IETF - IP Performance Metrics Registry). http://tools.ietf.org/wg/ippm/draft-ietf-ippm-metrics-registry/.

[14] The ElementTree iterparse Function. http://effbot.org/zone/element-iterparse.htm.

[15] B. Lowekamp, B. Tierney, L. Cottrell, R. Hughes-Jones, T. Kielmann, and M. Swany. A Hierarchy of Network Performance Characteristics for Grid Applications and Services. Community practice, Global Grid Forum, June 2003. http://nmwg.internet2.edu.

[16] Network Measurements Working Group (NM-WG). http://nmwg.internet2.edu.

[17] Performance focused Service Oriented Network monitoring ARchitecture. http://monstera.man.poznan.pl/wiki/index.php/Main_Page.

[18] Perl. http://www.perl.com/.

[19] The Python Programming Language. http://www.python.org/.
[20] RELAX-NG. http://www.relaxng.org/.
[21] Simple API for XML. http://www.saxproject.org/.
[22] Stanford Linear Accelerator Center. http://www.slac.stanford.edu/.
[23] RFC 1157, A Simple Network Management Protocol. http://www.ietf.org/rfc/rfc1157.txt.
[24] SOAP Specifications. http://www.w3.org/TR/soap/.
[25] StorCloud. http://www.vtksolutions.com/StorCloud/2005/.
[26] NM-WG schema and prototype repository. http://stout.pc.cis.ude.edu/NWMG/.
[27] Friedman's SuperSmoother. http://www.maths.lth.se/help/R/.R/library/modreg/html/supsmu.html.
[28] World Wide Web Consortium. http://www.w3.org/.
[29] Web Services. http://www.w3.org/2002/ws/.
[30] Web Services Notification. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn.
[31] Extensible Markup Language. http://www.w3.org/XML/.
[32] XML Schema language. http://www.w3.org/XML/Schema.
[33] Document Object Model. http://www.w3.org/DOM/.