GGF-NMWG-03-I
Network Measurements WG
http://www-didc.lbl.gov/NMWG/

D. Martin Swany
University of Delaware
Jason Zurawski
University of Delaware
Dan Gunter
Lawrence Berkeley National Lab
December 20, 2004

NMWG Schema Developers Guide

Status of this memo

This memo provides informtaion to the Grid community. It does not define any standards or technical recommendations. Distribution is unlimited

Copyright Notice

Abstract

We present a description and tutorial of the "normalized" schemas for Network Measurements

Contents

1 Introduction

1.1 Purpose

This document aims to help developers understand and use the Network Measurement working group's "normalized" schemas for measurement exchange.

1.2 Schema goals

The basic goal of the schema framework presented here is to allow separation of the variable information, which we call the "data", from the relatively constant information, which we call the "metadata". In the case of network measurements, an example of this separation would be between the "data" of the measurement values and timestamps, and the "metadata" of the network endpoints and measurement tool.

Slots for identifiers are provided to allow either the data or metadata to be referred to rather than always requiring it to be provided inline. This allows the data or metadata to be actually sent on the wire the minimum number of times, and referred to with a simple identifier thereafter.

2 Base schema

The base schema is more of a "framework schema", because it is intended to provide a general structure for concrete schemas such as the NM-WG network tool schemas. This section describes this schema.

2.1 Diagram

Figure 1 provides a diagram of the base schema.

Certain details in this figure have been rearranged for graphical simplicity. A more precise representation, using the RELAX-NG [1] "compact" syntax, may help to clarify.

2.2 Relax-NG Syntax

```
default namespace = "http://www.ggf.org/nmwg/"
namespace nmwg = "http://www.ggf.org/nmwg/"

start =
   element request { Request } |
   element response { Response }

Request =
  ## Properties of the request
  element properties { Properties }?,
  ## Request metadata (body of request)
  ## The request metadata could include an attribute for each element
  ## describing whether it requires and exact match, or whatever...
  Metadata*

Response =
  ## Properties of the response
  element properties { Properties }?,
   ## Metadata / data sections
   ( Metadata | Data ) *
```
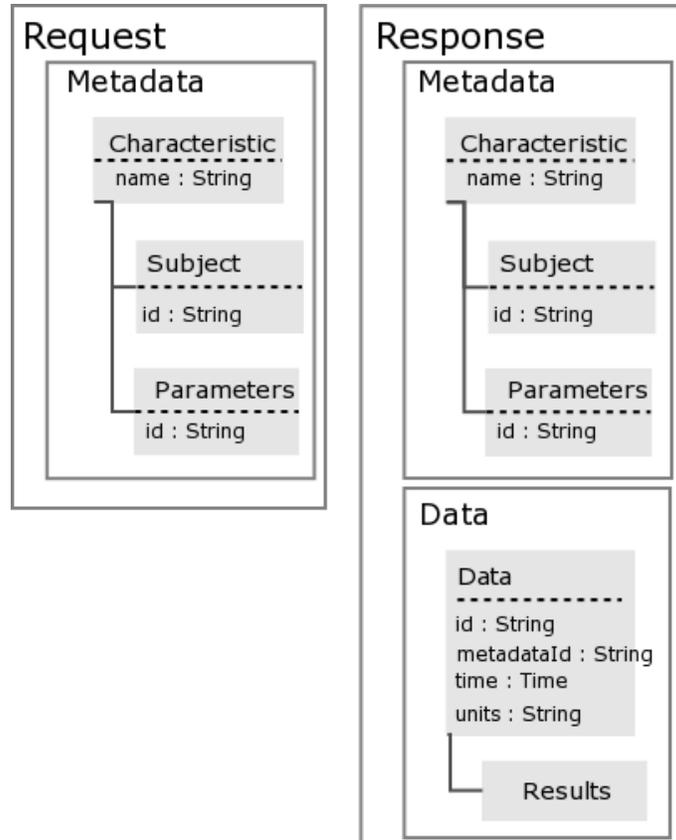
Figure 1: Base schema

```
## Metadata section for a request or response.
## This generic element will be replaced by
## a <metadata> element in some other namespace that
## identifies the actual type of metadata and
## constrains the type of the enclosed subject and
## parameters.
Metadata =
   element metadata {
      attribute id { Identifier },
      element subject { Subject },
      element parameters { Parameters }
   }

## Metadata subject
Subject =
```

```
  attribute id { Identifier }

## Metadata parameters
Parameters =
  attribute id { Identifier }

## Data section.
Data =
   element data {
      ## Data section identifier
      attribute id { Identifier },
      ## Reference to associated metadata section
      attribute metadataId { Identifier },
      ## Initial or "base" timestamp
      element time { Time }?,
      ## Units
      element units { token }?,
      ## Result data, subclasses will extend
      ## for their purposes
      Results
   }

## Base result type
Results =
   element results { text }

## Identifier for part of a request
## or response
Identifier = xsd:string

## Generic name/value pairs describing
## external properties of a request or
## response.
Properties =
   element item {
      ## Property name.
      attribute name { xsd:string },
      ## Identifier of request/response part
      ## that property refers to ("targets").
      ## If empty, then the entire request or response.
      attribute targetId { Identifier }?,
      ## Property value.
      text
   }+

## Timestamp
Time =
   ## Format and semantics of value
   attribute type { token },
   ## Timestamp value
   attribute value { text }
```

```
TimeRange =
   element start { Time },
   element end { Time | empty }
```

3 Basic Examples

3.1 Common elements

In addition to the base schemas, the examples below all share the basic topology elements needed for a host pair. The schema for these elements is shown below.

```
default namespace = "http://www.ggf.org/nmwg/"
namespace nmwg = "http://www.ggf.org/nmwg/"

HostPair =
   element hostPair {
      element src { Endpoint },
      element dst { Endpoint }
   }

HostPairQuery =
   ## Many different src/dst endpoints may match these patterns
   element hostPairQuery {
      element src { EndpointPattern }+,
      element dst { EndpointPattern }+
   }

Endpoint =
   ## 'hostname', 'ipv4', 'ipv6', other..
   attribute type { token },
   element address { string },
   element port { xsd:int }?

EndpointPattern =
   ## types for Endpoint as well as
   ## 'hostnameWildcard','ip{v4,v6}Wildcard', 'ip{v4,v6}Mask'
   ## e.g. '*.lbl.gov', 131.243.2.*, 131.243.0.0/16
   attribute type { token },
   element address { string },
   ## missing port implies 'any'
   element port { xsd:int }?
```

3.2 Traceroute

This section describes a NM-WG schema for requesting and reporting results from the "traceroute" tool.

- **Schema**

    ```
    namespace = "http://www.ggf.org/nmwg/traceroute/"
    namespace tr = "http://www.ggf.org/nmwg/traceroute/"
    ```

```
include "nmbase.rnc"
include "topology.rnc"

Metadata |=
    TracerouteMetadata

Results |=
    TracerouteResults

TracerouteMetadata =
    element tr:metadata {
        attribute id { Identifier },
        element tr:subject { TracerouteSubject },
        element tr:parameters { TracerouteParameters }
    }

TracerouteSubject =
    Subject,
    ( HostPair | HostPairQuery )

TracerouteParameters =
    Parameters,
    element maxttl { xsd:int }?,
    element nqueries { xsd:int }?
    # etc..

TracerouteResults =
    element tr:results {
        element probe {
            attribute num { xsd:int },
            element query {
                attribute num { xsd:int },
                element hopValue { string },
                element rtt { xsd:float }
            }
        }*
    }
```

- **Sample request and response**

```
<request xmlns:nmwg="http://www.ggf.org/nmwg/"
  xmlns="http://www.ggf.org/nmwg/"
  xmlns:tpgy="http://www.ggf.org/nmwg/topology/"
  xmlns:tr= "http://www.ggf.org/nmwg/traceroute/">
  <tr:metadata id="tr1">
    <tr:subject id="trs1">
      <tpgy:hostPairQuery>
        <tpgy:src type="ipv4">131.243.243</dst>
        <tpgy:dst type="hostnameWildcard">*.udel.edu</src>
      </tpgy:hostPairQuery>
    </tr:subject>
    <tr:parameters id="trp1">
```

```
        <tr:maxttl>32</tr:maxttl>
        <tr:nqueries>3</tr:nqueries>
      </tr:parameters>
    </tr:metadata>
 </request>

 <response xmlns:nmwg="http://www.ggf.org/nmwg/"
   xmlns="http://www.ggf.org/nmwg/"
   xmlns:tpgy="http://www.ggf.org/nmwg/topology/"
   xmlns:tr= "http://www.ggf.org/nmwg/traceroute/">
   <tr:metadata id="tr2">
     <tr:subject id="trs2">
       <tpgy:hostPair>
         <tpgy:src type="ipv4">131.243.243</dst>
         <tpgy:dst type="hostname">huey.udel.edu</src>
       </tpgy:hostPairQuery>
     </tr:subject>
     <tr:parameters id="trp2">
       <!--
              modified
       -->
       <tr:maxttl>30</tr:maxttl>
       <!--
              unchanged
       -->
       <tr:nqueries>3</tr:nqueries>
       <!--
              added
       -->
       <tr:dontFragment>0</tr:dontFragment>
       <tr:mode>UDP</tr:mode>
     </tr:parameters>
   </tr:metadata>
   <data id="trd1" metadataId="tr2">
     <time type="iso8601" value="2004-12-04T11:46:54.1132Z"/>
     <tr:results>
       <probe num='1'>
         <query num='1'>
           <hopValue>ir100gw-r2</hopValue>
           <rtt>0.233</rtt>
         </query>
         <query num='2'>
           <hopValue>ir100gw-r2</hopValue>
           <rtt>0.218</rtt>
         </query>
         <query num='3'>
           <hopValue>ir100gw-r2</hopValue>
           <rtt>0.195</rtt>
         </query>
       </probe>
       <!--
```

```
           ..skip to probe rejected hop #12
      -->
      <probe num='12'>
        <query num='1'>
          <hopValue/>
          <rtt/>
        </query>
        <query num='2'>
          <hopValue/>
          <rtt/>
        </query>
        <query num='3'>
          <hopValue/>
          <rtt/>
        </query>
      </probe>
      <!--
           ..skip to end
      -->
      <probe num='15'>
        <query num='1'>
          <hopValue>huey.udel.edu</hopValue>
          <rtt>79.099</rtt>
        </query>
        <query num='2'>
          <hopValue>huey.udel.edu</hopValue>
          <rtt>78.941</rtt>
        </query>
        <query num='3'>
          <hopValue>huey.udel.edu</hopValue>
          <rtt/>
        </query>
      </probe>
    </tr:results>
  </data>
</response>
```

3.3 Ping

This section describes a NM-WG schema for requesting and reporting results from the "ping" tool.

- **Schema**

  ```
  coming soon..
  ```

- **Sample request and response**

  ```
  coming soon..
  ```

3.4 Iperf

This section describes a NM-WG schema for requesting and reporting results from the "iperf" tool.

- **Schema**

  ```
  coming soon..
  ```

- **Sample request and response**

  ```
  coming soon..
  ```

4   Extended Examples

This section provides examples of some of multi-part exchanges, including data and metadata by reference rather than inline, etc.

4.1   Multi-part exchange

TBD

4.2   Include by reference

TBD

5   Extensibility

The base schema is designed to be extended. Moreover we envision that it will be extended by multiple parties simultaneously. In order to avoid conflicting extensions, we employ the most venerable of techniques: namespaces. This is directly supported by XML, so its application is straightforward. The rest of this section describes how one would extend the base schema to arrive at a derived domain-specific schema, and proposes a mechanism for standardizing extensions.

Let's say that, the owner of *www.foobar.biz*, wants to use the schema framework to exchange information about the number of failures of the "foo" system. They would follow these steps:

1. *Model the "foo" domain* To describe "foo" system information with the data/metadata framework, you would first separate the "foo" information into data and metadata, then further refine the metadata into characteristics (which can also be thought of as methods or operations), each of which associate with a type of subject and parameters.

   For example, let's say that each "foo" system has a simple name that can be easily encoded in a string. Therefore the subject needs to contain only that string. Previously, we said that the only characteristic we were interested in was number of failures. For this characteristic, then, all we need to do is define the parameters and expected structure of the results. Let's say that the parameters in this case is a simple time range (i.e., to answer questions like "how many failures last month?"). And the results are even simpler, just an integer number of failures.

2. *Choose a namespace* Now that the domain has been modeled, so to speak, the next step is to pick a unique namespace. This is shown in XML as a URI. To ensure uniqueness, you can pick some name that is centrally administrated and that you "own", i.e. your domain name. It is convenient to pick a resolvable URI that can lead to more information on your schema, so you choose a subdirectory of your site, "http://www.foobar.biz/schemas/foofailure/" as your namespace.

3. *Write the XML Schema* All that's left is the "dirty work" of codifying the structure that the XML should have. Usually, this is done by writing a schema in some XML schema language, such as RELAX-NG. In the RELAX-NG compact syntax, the foo system extension to the base schemas might look like this:

5.1 Foo schema extension

Declare the foo namespace.

```
namespace foo = http://www.foobar.biz/schemas/foofailure/
```

Define the foo metadata section, which is in that namespace, and includes subject and parameter sections also in the foo namespace. Because the subject and parameters elements are in this namespace, their child elements declared below are automatically also in this namespace, and so do not need the "foo:" prefix.

```
FooMetadata =
   element foo:metadata {
       attribute id { Identifier },
       element foo:subject { FooSubject },
       element foo:parameters { FooParameters }
   }
```

The foo subject is just a string. It extends the base schema subject, which contains just the identifier.

```
FooSubject =
   Subject,
   name { string }
```

The foo parameters are a time range. Like the foo subject, it extends the base schema parameters type to get the identifier for "free".

```
FooParameters =
   Parameters,
   element timeRange {
       element start { Time },
       element end { Time }
   }
```

Define a simple time stamp, used in the timeRange element above.

```
Time =
  attribute type { token },
  attribute value { text }
```

Now we just define the foo results to contain a single integer.

```
FooResults =
   element foo:results {
       element value { xsd:int }
   }
```

Finally, we need to combine all this by extending the base Metadata element with the FooMetadata and the base Results element (in the "data" section) with the FooResults.

```
Metadata |= FooMetadata
Results |= FooResults
```


5.2 Sample Foo request and response

We now have an extension that describes, and even "validate" in the XML-weenie sense of the word, foo failure requests and responses. An example XML exchange might be:

- **Request**

```
<request >
  <f:metadata id='100' xmlns:f='http://www.foobar.biz/schemas/foofailure/'>
    <f:subject id='110'>
      <name>Foo unit number 1</name>
    </f:subject>
    <f:parameters id='120'>
      <timeRange>
        <start type='iso8601' value='2004-11-01'/>
        <end type='iso8601' value='2004-11-30'/>
      </timeRange>
    </f:parameters>
  </f:metadata>
</request>
```

- **Response**

```
<response >
   <data id='1000' metadataId='100'>
      <time type='iso8601' value='2004-11-29T19:54:32.844197'/>
      <units>failures</units>
      <f:results xmlns:f='http://www.foobar.biz/schemas/foofailure/'>
        <value>3</value>
      </results>
   </data>
</response>
```

6 Use with Web Services

Many developers will be interested in using the schemas outlined above in a Web Services framework. This section provides some ideas for how to do this.

6.1 Overview

One way to build your web service is by starting with an interface file written in the Web Services Definition Language (WSDL) file, and writing your code to read and write XML documents that conform to that interface. We will call this approach "WSDL-first", as opposed to the "code-first" approach of programming classes that correspond to the data model and then serializing objects of those classes as web service invocations and responses.

Although it is certainly possible to use the concepts behind the NM-WG schemas with a "code-first" approach, this document does not provide any support for this. Specifically, we do not provide any particular language classes or APIs for exchanging NM-WG measurements. Instead, the highest layer we specify is the WSDL and the schemas for requests and responses that make up the bulk of that WSDL. In short, we only discuss the "WSDL-first" approach here.

In Web Services terminology, we are specifically supporting a "document/literal" style of invocation. What this means, basically, is that the Web Services layer provides only a very thin wrapper around the sending of one XML document and receipt of a second. In our case, these documents conform to the NM-WG request and response schemas, respectively. Current Web Services standards only support this simple request/response message pattern, so any more complicated exchange will need to be performed, for the time being, using application logic.

6.2 Approach

The recommended approach for deploying a web service using the NM-WG schemas is as follows:

1. *Determine desired functionality.* Select which parts of the open-ended universe of potential metadata the service should support and identify the relevant schemas.

2. *Decide on a validation approach* Validation of XML, which means not merely checking that every start tag has a matching end tag but also checking that the entire structure matches what is allowed by a schema, is optional. Sometimes, however, there are tools that can generate your code from the schemas and thus provide validation for "free". Java in particular has rich tools to perform this function, and there is an active effort in the Python community to provide similar (or better) functionality. Due to the complexity of this task, these tools are not perfect, so debugging the results has a certain learning curve. In addition, the mapping between XML structures and objects is not always intuitive.

   Programmatic validation, in other words checking "by hand" that the correct information is contained in the XML, is another option. This approach has the obvious drawback of requiring manual changes to the code every time the schemas and WSDL are updated. It has the potential drawback of not thoroughly checking the XML, although at times this may turn out to be an advantage by making the service less brittle to small changes in the schema.

   Some sort of validation approach has to be chosen, as it will guide the tools you use in future steps.

3. *Convert to XML Schema, if necessary* The *de facto* schema language is XML Schema, and most tools cannot deal with any other schema language. Despite this, all the schemas presented here are in a variant of a schema language RELAX-NG. Fortunately there are automatic conversion tools, such as Trang [2] and Sun's RELAX NG Converter [3].

4. *Combine schemas and a WSDL template*

   WSDL is a rather non-intuitive language, so it is easiest to start with a template for "document-oriented / literal" style interfaces, and just fill in the blanks.

   We provide here an example WSDL template. For modularity and reusability, we have created two `.wsdl` XML files, one containing all the schema definitions, and the other, "main", one that includes the first one and also provides details about where the service resides and how to contact it, called the *binding*. The idea is that the schema definitions file can be re-used with different concrete service instances, each of which will have different host and port information in the binding section.

   In the templates below, the "blanks" that need to be filled in are indicated like this: `*VARIABLE:sample value*` . Obviously, subsequent uses of the same variable omit the sample value.

   - **Schema definitions WSDL template**

     ```
     Filename: nmwg_definitions.wsdl

     <?xml version="1.0"?>
     <definitions name="*SCHEMA_NAME:NetworkMeasurement*"
       targetNamespace="*NM_NS:http://www.ggf.org/namespaces/2004/01/gridNetworkMonitori
       xmlns:tns="*NM_NS*/definitions"
       xmlns:nm="*NM_NS*"
       xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       xmlns="http://schemas.xmlsoap.org/wsdl/">

     <!-- Types -->
     ```

```
    <types>
      <xsd:schema targetNamespace="*NM_NS*">
          <xsd:include schemaLocation="*REQUEST_SCHEMA:request.xsd*" />
          <xsd:include schemaLocation="*REPORT_SCHEMA:response.xsd*" />
      </xsd:schema>
    </types>


    <!-- Messages -->

      <message name="NMWG-Request">
        <part name="body" element="nm:request"/>
      </message>

      <message name="NMWG-Response">
        <part name="body" element="nm:response"/>
      </message>

    <!-- Port types -->

    <portType name="NMWG-PortType">
        <operation name="NMWG-Get">
          <input message="tns:NMWG-Request" />
          <output message="tns:NMWG-Response" />
        </operation>
    </portType>

    </definitions>
```

- **Main/bindings WSDL template**

  Filename: nmwg_service.wsdl

```
  <?xml version="1.0"?>
  <definitions name="NetworkMeasurement"
    targetNamespace="*NM_NS*/service"
    xmlns:tns="*NM_NS*/service"
    xmlns:defs="*NM_NS*/definitions"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

  <!-- Import types, messages, port types -->
    <import
      namespace="*NM_NS*/definitions"
      location="nmwg_definitions.wsdl" />

  <!-- Bindings -->

    <binding name="NMWG-Binding"
      type="defs:NMWG-PortType">
      <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="NMWG-Get">
        <soap:operation
```

```
        soapAction="*HTTP_OPER:http://localhost/GetNetworkMeasurement*"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>

  <!-- Service -->

  <service name="NMWG-Service">
    <documentation>
      Grid NM-WG Network Measurement service
    </documentation>
    <port name="NMWG-Port"
      binding="tns:NMWG-Binding">
      <soap:address location="*SVC_URLhttp://localhost:8000*"/>
    </port>
  </service>
</definitions>
```

5. *Create web service code (client and/or server)*

   Depending on the requirements, client and/or server code should now be written. There are tools in Java and Python to generate code stubs from WSDL documents that handle most of the messy details of SOAP invocation, exposing the XML data to the user as hierarchies of objects and attributes. Manual approaches that directly manipulate the XML, as well as many variations in between, are possible. As long as the WSDL is still the canonical interface definition, interoperability should still be possible even though communicating parties have taken different approaches.

6.3   Have a beer

By now you're probably in need of one. But don't stoop to that weak swill that is factory produced in Milwaukee – that stuff has corn and rice in it. Get something made from barley, yeast and hops, pour yourself a glass of it, and try to remember a time before Web Services and XML Schema. A kinder, gentler time...

References

[1] James Clark. RELAX-NG home page. http://www.relaxng.org.

[2] James Clark. Trang home page. http://thaiopensource.com/relaxng/trang.html.

[3] Kohsuke Kawaguchi.   Sun RELAX-NG converter.   http://wwws.sun.com/software/xml/developers/relaxngconverter/.