GWD-R
Network Measurement Control Working Group
https://forge.gridforum.org/projects/nmc-wg

Jason Zurawski, Internet2
Martin Swany, University of Delaware
January 19, 2012

An Extensible Protocol for Network Measurement and Control DRAFT

Status of This Document

This document provides information to the Grid community regarding the design of protocols to control software engaged in the creation, storage, and exchange of network measurements. Distribution is unlimited.

Copyright Notice

1 Abstract

The exchange of network measurement and performance data is a common problem for communities that rely on distributed computing methods such as the Grid and dynamic provisioning of network circuits. Work produced in related efforts, such as the Network Measurements Working Group (**NM-WG**) [8], has provided an extensible mechanism for the *representation* of these data sets but does not provide guidance for *collection*, *storage*, or *exchange*.

The Network Measurement Control Working Group (**NMC-WG**) [6] has taken the task of designing flexible *protocols* to be implemented by software systems that are interested in working with network measurements as encoded by the **NM-WG** standards. These protocols provide the basic communication recommendations that a service should implement. The recommendations brought forward in this work should not reflect any one implementation: these protocols are kept general and are extensible to many use cases and potential software systems.

Contents

2 Introduction

This document describes a *protocol* designed to manage the interaction between systems focused on the *collection*, *storage*, and *exchange* of network performance information. This protocol becomes relevant in an incarnation of a performance measurement framework; consisting of a set of services, each acting as an intermediate layer, between the performance measurement tools and the diagnostic or visualization applications all within a federated environment. This specification defines a service-oriented design style with isolated a set of functions so that each function can be delivered by potentially different software entities.

In this model, all services must communicate using well-defined protocols. While this document does not formally specify these details, it is envisioned that implementations of the proposed architecture shall make this functionality available via *Web Services* (WS). Existing implementation either use SOAP over HTTP or RESTful webservices, but no requirement is given in this document for a particular transport protocol.

The work presented here builds upon the output of other working groups focused on the accurate description of network measurements [8] and topological representation of network elements [7]. When applicable, we will directly cite terminology and ideas from these working groups. We do not describe a particular system currently in use, although several prototypes exist that implement messaging similar to this work.

3 Motivation

A common message exchange pattern for Web Services (WS) is a *Request* followed by a *Response*. This particular communication pattern is important and involves two actors for the general case. Consider the example in Figure 1 of a *client* application interacting with some service, in this case a *measurement service*. This exchange assumes that both the client and the service speak a common dialect of some communication protocol.

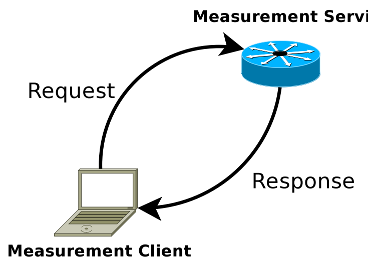

Figure 1: Two serial two way exchanges between a client and a service. The middle actor acts both as a client and a server.

It becomes the burden of both the service developer and the client developer to document what an exchange consists of as well as how it takes place (albeit from different points of view). A second example pictured in Figure 2 illustrates that a single actor can be involved in multiple message exchanges, with different roles.

Another interaction is the notion of a *Subscription*, *Notification* or other form of “one-sided” exchange. Services may use this to perform tasks such as subscribing to status updates from a service, or otherwise

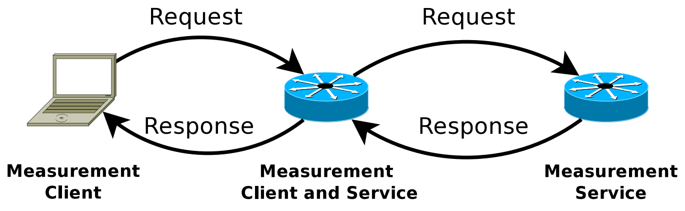

Figure 2: Simple two way exchange between two services.

alerting a service or client about the existence of some key piece of information. Figure 3 describes this exchange in detail.

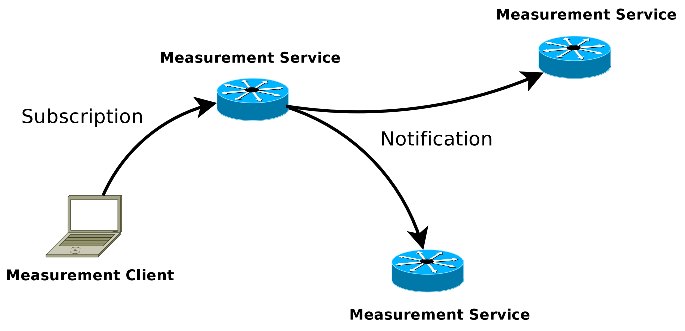

Figure 3: One way exchange where a client subscribes to a service. Other services may already receive notifications.

The acts of sending *requests*, receiving *responses*, and being able to discern success or failure are common across many specific interactions. One aim of this document is to prevent the following redundancies:

- **Duplicate Schemata -** *Messages* differ from service to service, but the overarching concepts will not; we present some of the common features that must be present in every exchange and describe how extension is possible.
- **Duplicate Semantic Principals -** Concepts used in services designed in the past are carried over into future iterations. Common practices and common features are only described once.
- **Duplicate Error Conditions -** Some errors will occur across services and are only defined once (e.g. *Unknown Message Type*).
- **Duplication of Common Exchanges -** Common protocols must be the same among services in a framework. For example being able to retrieve the "status" of a neighbor or wrapping your communication in a secure channel are communication patterns that will be required everywhere. A common *Echo* protocol is presented in this document as an example.

With a set design and format to this base functionality, it is possible to define protocol extensions on a "type by type" basis instead of a "service by service" approach. This also allows for a sufficient reduction in documentation due to service types implementing the same underlying *format* of messages (e.g. services that store measurements may implement the same message types with notable exceptions and data differences).

It is expected that the following extensions to this base document will be prepared to describe specific service interactions:

- **Measurement Collection -** Describes the process of communicating with services that perform and gather measurements
- **Measurement Storage -** Describes communication with services that archive measurements
- **Information Location -** Interacting with services that index and locate services and data
- **Service Authentication -** Describes communication within secure data channels in a federated environment

4 Messages

As described in Section 3, the communication protocol is simple and based on the notion of *Request* and *Response* messages. The construction of the *message* itself takes advantage of work produced in the **NM-WG** by re-using several key constructs. Both message types are fundamentally the same: a series of *metadata* and *data* units linked via identifying attributes (e.g. *id* and *idRef* attribute values). These concepts, shown in [10, 13], observe the same rules with regards to splitting both measurement and communication.

4.1 Preliminary Example

To cement an early understanding of how the messages work, consider this rudimentary query for some data from a service:

```
<message type="request">

  <metadata id="m1">
    <!-- some partial metadata the service may or may not understand -->
  </metadata>

  <data id="d1" metadatIdRef="m1" />

</message>
```

There are some important things to note about this query:

- **Message Type -** Every *message* must contain a *type*, these facilitate the semantic intentions of the internal data
- **Message Structure -** There must be *metadata* and/or *data* elements in each message; there does not need to be a matching data for every metadata (e.g. *Chaining*, see Section 5)
- **Metadata -** Must contain measurement data, identifying information about a service, or even information meant to serve as a modifier (see see Section 5). Note that we still loosely observe the static rule of thumb
- **Data -** Serves a dual role: in request messages this may be empty (e.g. this is a *data trigger*. An empty data element lets the service know we need action on the accompanying metadata), or it may contain dynamic measurement data or even other metadata elements

Simply stated, we are sending a request (either partial or complete) to a capable service to perform some interactive behavior. We are interested in having the service verify that it is able to service our request, either by acting in a positive or negative manner. We are interested in either setting or retrieving information on this target data set - subsequent operations may be necessary. When the service receives this request it will check for several things:

- **Syntax -** Does the request parse correctly? Incorrect syntax will trigger *error routines*.
- **Message Type -** Can this service accept and act on this kind of message? An unexpected message must be rejected outright by discarding it and responding with a corresponding error message.
- **Structure -** Is there at least a single metadata and data pair that is capable of being acted on? A service that cannot determine if there is actionable content in the request will *discard* the message with an *error routine*.
- **Semantics -** Does the request make sense according to the schematic rules; can the metadata be acted upon; are the chains resolved properly? Handling of semantic rules are the discretion of the service: *error routines* are possible or perhaps some form of *panic parsing* may result in partial completion of a request.

The service has two options at this point: acting on the message (e.g. returning data) or reporting some other form of "status" (e.g. an *error* message). We will explore the first option initially:

```
<message type="response">

  <metadata id="m1">
    <!-- specific metadata that matched -->
  </metadata>

  <data id="d1" metadatIdRef="m1">
    <!-- data, or perhaps a pointer (key) to data -->
  </data>

</message>
```

Note this message is similar to the request in many ways. The major differences:

- **Message Type -** This becomes the foil of the previous request; it is common to simply replace the word "Request" with "Response"
- **Message Structure -** All valid metadata and data pairings must be acted on, chained items may be truncated
- **Metadata -** May be "completed" (e.g. augmented information added to the original) if it was incomplete in the initial request
- **Data -** Must contain information, especially if it was empty in the initial request

The second situation is not very different, but is indicative of something occurring that was not expected. We don't explicitly use the term "error" to describe this situation because many paths that lead to this are not *wrong*. Some examples of status may be:

- **Message Syntax/Structure/Semantics -** The service must be able to understand the request, if it cannot be parsed on either the syntactic or semantic levels the status must reflect this.

- **Metadata/Data Search -** The backend storage may simply be devoid of references to what a request is interested in, this should be expressed by returning nothing in the response or having an explicit message to do so.

- **Catastrophic Events -** Internal events may trigger some sort of panic in the service; note that not all events may be recovered from and are not the fault of the service itself (host machine failures, etc.).

The general format of a status message is as follows, parallels between the previous response as well as the request can easily be drawn.

```
<message type="response">

  <metadata id="m1">
    <!-- some (machine level) status information -->
  </metadata>

  <data id="d1" metadatIdRef="m1">
    <!-- some human readable status information -->
  </data>

</message>
```

4.2 Message Actions

The examples from Section 4.1 characterizes many of the common actions services perform on receipt of a request message. A more formal description of this interaction is described in Figure 4. Note that this is a generalized attempt, and should not directly reflect the actions of any particular service. Protocol extensions should provide a description where this example is lacking.

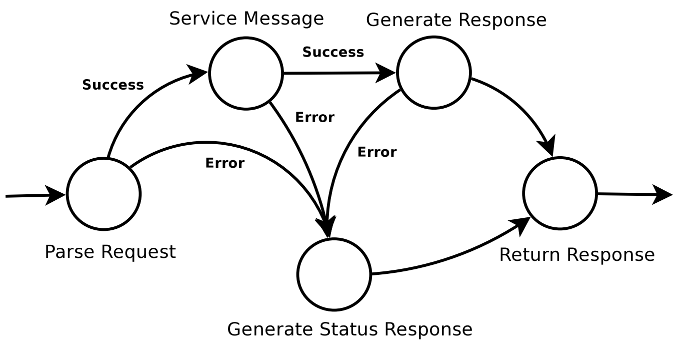

Figure 4: Typical programmatic flow of a messages through a measurement service.

The example illustrates that any stage of processing a request may trigger entry into the status routine. Specifics regarding available status messages are available in Section 6.

4.3 Request Message

The *Request Message* is a container for submitting communication to capable services. Enclosed in this simple envelope must be a series of *metadata* and *data* pairs containing various instructions to act out. We first present a very simple schema in Section 4.3.1 along with an analysis of the elements in Section 4.3.2. We conclude with examples in Section 4.3.3.

4.3.1 Request Message Schema

The following schema is a native description of the request schema as in the RELAX-NG[9] language. Through the use of tools such as Trang[11] and MSV[5] it is possible to convert this to other widely accepted formats such as XSD[12].

```
# Begin Schema

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

start =
  element nmwg:message {
    Identifier? &
    MessageIdRef? &
    attribute type { "Request" } &
    Parameters? &
    (
      Metadata |
      Data
    )+
  }

Parameters =
  element nmwg:parameters {
    Identifier &
    Parameter+
  }

Parameter =
  element nmwg:parameter {
    attribute name { xsd:string } &
    (
      attribute value { xsd:string } |
      (
        anyElement |
        text
      )
    )
  }

Metadata =
  element nmwg:metadata {
    Identifier &
    MetadataIdentifierRef? &
    anyElement*
  }

Data =
  element nmwg:data {
    Identifier &
    MetadataIdentifierRef &
    anyElement*
  }

Identifier =
  attribute id { xsd:string }

MessageIdRef =
  attribute messageIdRef { xsd:string }

MetadataIdentifierRef =
  attribute metadataIdRef { xsd:string }

anyElement =
  element * {
    anyThing
  }

anyAttribute =
  attribute * { text }

anyThing =
  (
    anyElement |
    anyAttribute |
    text
  )*


# End Schema
```

4.3.2 Request Message Analysis

The following is a breakdown of the elements featured in the schema. Note that services in general must not implement or attempt to understand this, it is provided as a tool to aid in the development of extensions.

4.3.2.1 Message

```
<nmwg:message xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
              id="message1"
              type="Request">

  <nmwg:parameters />

  <nmwg:metadata />

  <nmwg:data />

</nmwg:message>
```

Table 1: Message Element Specifics

Message Element	
localname	message
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id, messageIdRef, type
nested elements	parameters, metadata, data
required	yes

The message construct is meant to serve as a container for transporting *requests* to capable services. The message element itself is unremarkable, it features *attributes* to aid in the identification of messages (e.g. *id*s) and contains elements with measurement or instructional content. We first examine the available attributes:

- **id -** Identifier that may be used to track state between messages and services
- **messageIdRef -** Optional identifier that may be used to track state to previous message exchanges.
- **type -** Must designate the message to a particular type; fully enumerated in each protocol extension

There are three major elements that should be contained in the message element:

- **Parameters -** Described in Section 4.3.2.2
- **Metadata -** Described in Section 4.3.2.4
- **Data -** Described in Section 4.3.2.8

4.3.2.2 Parameters

```
<nmwg:parameters xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="parameters1">

  <nmwg:parameter />

</nmwg:parameters>
```

Table 2: Parameters Element Specifics

Parameters Element	
localname	parameters
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id
nested elements	parameter
required	no

The parameters element encloses a series of parameter elements that may be used to adjust variable aspects of this schema. This element serves as a container for the *Parameter* (see Section 4.3.2.3) elements that must populate it. The single available attribute is described first:

- **id -** Identifying attribute that may be used to track state.

The element (only one possible in this case) is described next:

- **Parameter -** Described in Section 4.3.2.3

Note that the use of this element (in this particular location) is optional. Services are not required to understand this element and should ignore this element if not expected by the service. Please consult service documentation before proceeding.

4.3.2.3 Parameter

```
<nmwg:parameter xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
               name="NAME">VALUE</nmwg:parameter>

<!-- OR -->

<nmwg:parameter xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
               name="NAME" value="VALUE" />
```

Table 3: Parameter Element Specifics

Parameter Element	
localname	parameter
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	name, value
nested elements	text, undefined
required	yes

The parameter element features a generic structure that allows easy adaptation to the needs of a particular schema. For brevity, possible names and values are not listed here and are beyond the scope of this document. This exercise must be done at the protocol extension and service documentation level.

- **name -** Must specify the name of some variable value

- **value -** May be used instead a text element (or enclosed element) to set the value of the *name*

In lieu of the *value* attribute, a *text* (or unspecified *complex*) element may be used for the same purpose. It is recommended that protocol extensions adopt a single method for all uses of this element. The other possibility for element containment is left unspecified. We do not rule out that "alternate" elements would be useful in this case, but the exact use is left up to other extensions.

4.3.2.4 Metadata

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
               id="metadata2" metadataIdRef="metadata1">

  <!-- Possible Values Include: -->

  <nmwg:subject />

  <nmwg:key />

  <nmwg:eventType />

  <nmwg:parameters />

</nmwg:metadata>
```

Table 4: Metadata Element Specifics

Metadata Element	
localname	metadata
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id, metadataIdRef
nested elements	undefined, (*subject*, *key*, *eventType*, and *parameters* are common)
required	yes

The metadata element normally contains the static parts of measurements, and shall differ from service to service. Besides measurement data it is possible to send other items such as *service descriptions*. The schema description itself of what is possible inside of this element uses vague language that allows for *any* reasonable XML to be contained within. The most common elements that are included are *Subject* (see Section 4.3.2.5), *Key* (see Section 4.3.2.6), *EventType* (see Section 4.3.2.7), and *Parameters* (see Section 4.3.2.2). We will present only a brief discussion of these within this document; a more exact definition should be found in specific measurement documentation.

There are two attributes possible. These should be used to both track state and perform the various forms of chaining (e.g. *operator* or *merge*) that a request message may require. A detailed description of this element follows:

- **id -** Identifying attribute that may be used to track state.

- **metadataIdRef -** Identifying attribute that may be used to track state or for *chaining* procedures.

4.3.2.5 Subject

```
<nmwg:subject xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
              id="subject1" metadataIdRef="metadata1" />
```

Table 5: Subject Element Specifics

Subject Element	
localname	subject
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id, metadataIdRef
nested elements	undefined, (topology elements are common)
required	N/A

The subject element normally contains *topological* specifications that relate directly to a measurement or a specific service. We leave a full description of this element up to individual implementations but mention it here due to common use. There are two recommended attributes, these are used to both track state and perform a specific type of *chaining* (e.g. *subject* chaining) that may be required in a request message. A detailed description follows:

- **id -** Identifying attribute that may be used to track state.
- **metadataIdRef -** Identifying attribute that may be used to track state or used in *chaining*.

4.3.2.6 Key

example of status response in 4.1 does not explain too much (looks the ¿ same as earlier response example)

```
<nmwg:key xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/">
  <nmwg:parameters />
</nmwg:key>
```

Table 6: Key Element Specifics

Key Element	
localname	key
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id
nested elements	parameters
required	N/A

The key element is rooted in the concept of a *hash function* — a function or process designed to convert a variable amount of information into a single value or *index*. Once converted this single value can then be used a shorthanded notation to reference the original entity, imparting a performance increase for computational tasks.

The key structure shall used to convey sensitive or private information to and from the service. For this reason the contents of the key must be viewed as "opaque", and must not be dissected. The key should contain a *Parameters* (see Section 4.3.2.2) element. There is only one attribute possible: *id*. This may used to track state. A detailed description follows:

- **id -** Identifying attribute that may be used to track state.

4.3.2.7 EventType

```
<nmwg:eventType xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/">TEXT</nmwg:eventType>
```

Table 7: EventType Element Specifics

EventType Element	
localname	eventType
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	N/A
nested elements	text
required	N/A

The eventType element must used to describe a measurement's specific data type (e.g. closely matching the definitions described in [2] and [3]) or should be used to trigger an internal event within the service. This element contains no attributes, and must only contain text, normally in the form of a *URI*. There may be *many* eventType elements in a single metadata.

4.3.2.8 Data

```
<nmwg:data xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
           id="data2" metadataIdRef="metadata2" />
```

Table 8: Data Element Specifics

Data Element	
localname	data
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id, metadataIdRef
nested elements	undefined
required	yes

The data element must contain the dynamic parts of measurements, and shall differ from service to service. Besides collected measurements the data field may also be populated with query data, or even other other metadata information in certain applications. We leave the description of what is possible inside of *data* blank, and use vague schema language that allows for *any* reasonable content to be contained within.

There are two attributes possible. These may used to track state inside of a request message. A detailed description follows:

- **id -** Identifying attribute that may be used to track state.
- **metadataIdRef -** must be used to link data to metadata.

4.3.3 Request Message Example

The following examples demonstrate some of the possible uses and layouts of request messages in the base protocol. These examples are not an attempt to be exhaustive, but rather some examples of ways to perform common tasks. Note that these messages are not indicative of a particular service.

The first example demonstrates the most common use case: a single metadata and data pair. This message represents the layouts of most request messages in a generic measurement framework.

```
<!-- Begin XML -->

<message xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
         type="request"
         id="message1">

  <metadata id="m1">
    <!-- metadata -->
  </metadata>

  <!-- data trigger -->
  <data id="d1" metadataIdRef="m1" />

</message>

<!-- End XML -->
```

The second example is similar, but incorporates a parameters block that may be populated with optional behaviors of a service.

```
<!-- Begin XML -->

<message xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
         type="request"
         id="message2">

  <nmwg:parameters id="parameters1">
    <nmwg:parameter name="name">value</nmwg:parameter>
  </nmwg:parameters>

  <metadata id="m1">
    <!-- metadata -->
  </metadata>

  <!-- data trigger -->
  <data id="d1" metadataIdRef="m1" />

</message>

<!-- End XML -->
```

The third example is also similar to the first, but shows it is possible to ask for multiple pairs of metadata and data in a single message. Note that there are two empty data triggers to signify that each message be acted upon.

```
<!-- Begin XML -->

<message xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
         type="request"
         id="message3">

  <metadata id="m1">
    <!-- metadata -->
  </metadata>

  <!-- data trigger -->
  <data id="d1" metadataIdRef="m1" />

  <metadata id="m2">
    <!-- another metadata -->
  </metadata>
  <!-- data trigger -->
  <data id="d2" metadataIdRef="m2" />

</message>

<!-- End XML -->
```

This example features merge chaining. Note there is only one data trigger, and it is at the tail of the chain. A service would perform the necessary chaining first, then act on the result of this operation.

```
<!-- Begin XML -->

<message xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
         type="request"
         id="message4">

  <metadata id="m1">
    <!-- more metadata -->
  </metadata>

  <metadata id="m2" metadataIdRef="m1">
    <!-- metadata -->
  </metadata>

  <!-- data trigger -->
  <data id="d1" metadataIdRef="m2" />

</message>

<!-- End XML -->
```

The final example is an invalid case where the metadata does not have an appropriate data trigger.

```
<!-- Begin XML -->

<message xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
         type="request"
         id="message5">

  <metadata id="m1">
    <!-- metadata -->
  </metadata>

  <!-- data trigger -->
  <data id="d1" metadataIdRef="m2" />

</message>

<!-- End XML -->
```

4.4 Response Message

The response message is a container filled with the results of a *Response Message* from a capable service. Enclosed in this simple envelope must be a series of metadata and data pairs containing the results of actions performed by a service. We first present a very simple schema in Section 4.4.1 along with an analysis of the elements in Section 4.4.2. We conclude with examples in Section 4.4.3.

4.4.1 Response Message Schema

The following schema is a native description of the request schema as in the RELAX-NG[9] language. Through the use of tools such as Trang[11] and MSV[5] it is possible to convert this to other widely accepted formats such as XSD[12].

```
# Begin Schema

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

start =
  element nmwg:message {
    Identifier? &
    attribute messageIdRef { xsd:string }? &
    attribute type { "Response" } &
    Parameters? &
    (
      Metadata |
      Data
    )+
  }

Parameters =
  element nmwg:parameters {
    Identifier &
    Parameter+
  }

Parameter =
  element nmwg:parameter {
    attribute name { xsd:string } &
    (
      attribute value { xsd:string } |
      (
        anyElement |
        text
      )
    )
  }

Metadata =
  element nmwg:metadata {
    Identifier &
    MetadataIdentifierRef? &
    anyElement*
  }

Data =
  element nmwg:data {
    Identifier &
    MetadataIdentifierRef &
    anyElement*
  }

Identifier =
  attribute id { xsd:string }

MetadataIdentifierRef =
  attribute metadataIdRef { xsd:string }

anyElement =
  element * {
    anyThing
  }

anyAttribute =
  attribute * { text }

anyThing =
  (
    anyElement |
    anyAttribute |
    text
  )*


# End Schema
```

4.4.2 Response Message Analysis

The following is a breakdown of the elements featured in the schema. Note that services in general must not implement or attempt to understand this, it is provided as a tool to aid in the development of extensions.

4.4.2.1 Message

```
<nmwg:message xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
              id="message2"
              messageIdRef="message1"
              type="Response">

  <nmwg:parameters />

  <nmwg:metadata />

  <nmwg:data />

</nmwg:message>
```

Table 9: Message Element Specifics

Message Element	
localname	message
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id, type, messageIdRef
nested elements	parameters, metadata, data
required	yes

The message element, like it's counterpart seen in Section 4.3.2.1 serves as a container for transporting responses from capable services. The message itself is unremarkable, it features attributes to aid in the identification of messages and contains elements with measurement or instructional content. We first examine the available attributes:

- **id -** Identifier that may be used to track state between messages and services
- **type -** Must designate the message to a particular type; fully enumerated in each protocol extension
- **messageIdRef -** Identifier that may be used to track state between messages and services

There are three major elements that may be contained in the message element:

- **Parameters -** Described in Section 4.4.2.2
- **Metadata -** Described in Section 4.4.2.4
- **Data -** Described in Section 4.4.2.5

4.4.2.2 Parameters

```
<nmwg:parameters xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="parameters1">

  <nmwg:parameter />

</nmwg:parameters>
```

The parameters element should only be present in the message element if there was a corresponding element in the *Request Message* (see Section 4.3.2.1). It may also be used by services to relay back other forms of information. As in Section 4.3.2.2, it encloses a series of *parameter* elements. This element serves merely as a container for the Parameter elements (see Section 4.4.2.3) that will populate it. The single available attribute is described first:

Table 10: Parameters Element Specifics

Parameters Element	
localname	parameters
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id
nested elements	parameter
required	no

- **id -** Identifying attribute that may be used to track state.

There is only one available element, although it may be used multiple times

- **Parameter -** Described in Section 4.4.2.3

4.4.2.3 Parameter

```
<nmwg:parameter xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
                name="NAME">VALUE</nmwg:parameter>

<!-- OR -->

<nmwg:parameter xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
                name="NAME" value="VALUE" />
```

Table 11: Parameter Element Specifics

Parameter Element	
localname	parameter
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	name, value
nested elements	text, undefined
required	yes

The parameter element features a generic structure that allows it to easily adapt to the needs of a particular schema. For brevity, possible names and values are not listed here and are beyond the scope of this document. This exercise must be done at the protocol extension and service documentation level.

- **name -** Must generically specify the name of some variable value
- **value -** May be used instead a text element to set the value of the *name*

In lieu of the *value* attribute, a text element may be used for the same purpose. It is recommended that protocol extensions adopt a single method for all uses of this element. The other possibility for element containment is left unspecified. We do not rule out that many elements would be useful in this case, but the exact use is left up to other extensions.

4.4.2.4 Metadata

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
               id="metadata2" metadataIdRef="metadata1">

  <!-- These elements are commonly used: -->

  <nwmg:subject />

  <nmwg:key />

  <nmwg:eventType />

  <nmwg:parameters />

</nmwg:metadata>
```

Table 12: Metadata Element Specifics

Metadata Element	
localname	metadata
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id, metadataIdRef
nested elements	undefined, (subject, key, eventType, and parameters are common)
required	yes

The metadata element in the response is normally an exact copy of the sent *Metadata* (see Section 4.3.2.4). We leave the description of what is possible inside of a metadata blank, and use vague schema language that allows for any reasonable XML to be contained within.

There are two attributes possible. These may be used to track state, possibly back to the sent *Metadata*. A detailed description follows:

- **id -** Identifying attribute that may be used to track state.
- **metadataIdRef -** Identifying attribute that may be used to track state.

4.4.2.5 Data

```
<nmwg:data xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
           id="data2" metadataIdRef="metadata2">

  <nmwg:datum />

  <nmwg:key />

  <nmwg:metadata />

</nmwg:data>
```

The data element will contain results and is usually not not empty like the trigger that is used in *Data* (see Section 4.3.2.8). We leave the description of what is possible inside of a data blank, and use vague schema language that allows for any reasonable XML to be contained within.

There are two attributes possible. These may be used to both track state inside of a response message. A detailed description follows:

- **id -** Identifying attribute that may be used to track state.
- **metadataIdRef -** Must be used to link data to metadata.

Table 13: Data Element Specifics

Data Element	
localname	data
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id, metadataIdRef
nested elements	undefined, (datum, key, and metadata are common)
required	yes

4.4.2.6 Key

```
<nmwg:key xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/">
  <nmwg:parameters />
</nmwg:key>
```

Table 14: Key Element Specifics

Key Element	
localname	key
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id
nested elements	parameters
required	no

The key structure should be used to convey sensitive or private information to and from the service. For this reason the contents of the key must be viewed as opaque, and generally not be dissected. The key should contain the *Parameters* (see Section 4.4.2.2) element. There is only one attributes possible: id. This may be used to track state. A detailed description follows:

- **id -** Identifying attribute that may be used to track state.

4.4.2.7 Datum

```
<nmwg:datum xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" />
```

The datum element should be used to relay various types of information. Common uses are to return measurement observations (e.g. time and value pairs) or even status (e.g. error messages). We leave the attributes and nested elements purposely undefined as they may differ in various profiles of this document.

4.4.3 Response Message Example

The following examples demonstrate some of the possible uses and layouts of response messages in the base protocol. These examples are not an attempt to be exhaustive and are not indicative of a particular service.

The first example is the most common form of response message containing a single metadata and data pair. This would be indicative of success.

Table 15: Datum Element Specifics

Datum Element	
localname	datum
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	undefined
nested elements	undefined
required	no

```
<!-- Begin XML -->

<message xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
         type="response"
         id="message1"
         messageIdRef="someothermessage">

  <metadata id="m1">
    <!-- metadata -->
  </metadata>

  <data id="d1" metadataIdRef="m1">
    <!-- datum stuffs -->
  </data>

</message>

<!-- End XML -->
```

The second example is similar, although it features two pairs.

```
<!-- Begin XML -->

<message xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
         type="response"
         id="message1"
         messageIdRef="someothermessage">

  <metadata id="m1">
    <!-- metadata -->
  </metadata>

  <data id="d1" metadataIdRef="m1">
    <!-- datum stuffs -->
  </data>

  <metadata id="m2">
    <!-- another metadata -->
  </metadata>

  <data id="d2" metadataIdRef="m2">
    <!-- datum stuffs -->
  </data>

</message>

<!-- End XML -->
```

The final example demonstrates an error condition. Note that this may contain multiple pairs if sent, and it may be possible to have success for some, and errors for others.

```
<!-- Begin XML -->

<message xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
         type="response"
         id="message1"
         messageIdRef="someothermessage">
```

```
  <metadata id="m1">
    <!-- some error -->
  </metadata>

  <data id="d1" metadataIdRef="m1">
    <!-- some error message -->
  </data>

</message>

<!-- End XML -->
```

5 Information Chaining

Since inception, a key goal of the protocols has been extension. The authors of the original schemata realized that not every situation could easily be described though the basic constructs; extending the basic building blocks to complex situations is paramount. Uncharted concepts could be represented with newly created constructs each time a foreign abstraction came to light; but extension and backwards compatibility must be favored over quick and easy solutions. Therefore, basic extension mechanisms, known as chaining, are the recognized procedure to extend metadata constructs as well as express other operations on the underlying data.

This section presents the major uses of chaining; note that individual service implementations may choose to strictly or loosely interpret these guidelines for the sake of performance or protection. The protocol itself offers no specific guidance on these issues in favor of simply describing the structural composition of both the input data and the resulting output.

Chaining itself has taken on two major forms: *merge chaining* described in Section 5.1 and *filter chaining* described in Section 5.2. These two instances will be described first in broad terms that explain the logic and reasoning of why each operation makes sense, and in what context they should be employed. The specific syntax and transformation steps will be presented in the next section.

5.1 Merge Chaining

As the name implies, we intend to *merge* or combine metadata elements through this structure. There are many things we may consider when describing this operation:

- Which elements are *mergeable*?
- How much *recursion* is needed for merge-able elements?
- When should we *duplicate* elements?
- When should we *replace* elements in the course of merging?

As stated previously, the schemata itself does not offer any suggestions as to what is a *good merge* vs. a *bad merge*. There are no rules regarding which *types* of data should and should not be merged. There is no guidance on when we should duplicate or replace elements.

We recommend some very simple and succinct guidelines that services may implement for this particular style of merging. There shall be exceptions to rules, therefore the reader is encouraged to think carefully about what a specific service may need when implementing this recommendation.

5.1.1 Mergeable Elements and Recursion

When merging we must first look at the *top-level* elements; namely subject, eventType, and parameters. When faced with two metadata blocks to be merged, we only wish to combine:

- *Like* Elements (e.g. sharing the same localname)
- Elements in the same namespace
- Elements sharing the same (or "similar") eventType

When this first criteria is met, we must traverse the chain *downward*, recursively. A clear question to answer is "How far should we venture into the XML structure looking for similarities or differences"? This question does not have a definite answer such as "Stop at the grandchild of the current element". While this may be frustrating, domain knowledge shall help you make a passable decision especially with regards to topology based elements.

Like elements that do not share a common namespace will require special rules that may differ from service to service. Depending on the level of protection or speed we wish to attain, these rules may vary. Service and protocol documentation must fill in details beyond the scope of this work.

5.1.2 Duplication, Augmentation, and Replacement

When are faced with *like* elements that do not share a common namespace, we should not combine. We must try to find the *least significant* namespace and work from there. Additionally we may run into items that are *exactly* the same (such as certain *parameters*, or *eventTypes*). In some cases we should take care to *add* all of these together to make duplicates; other cases may dictate total replacement. Specific rule such as these are best left to a service designer.

As an example of extreme cases, consider taking a very safe approach to the combining of elements (i.e. not merging *like* elements with different namespaces). This approach will ensure that we protect the schema differences but may result in many more *wrong* answers when it comes to searching. The converse is a very dangerous approach where we merge items that could be different on the inside. This may result in an approach similar to *I know what you meant* and could yield a more robust query mechanism (providing intuitive answers when something may not completely match, rejecting outright things that do not make sense).

5.1.3 Merge Chaining Examples

A classic example of merge chaining is to partially specify a metadata (leaving out perhaps one unspecified element) and then constructing new elements from this original. This example does not feature any *overwriting* of duplicate elements.

Take for example a physical *Layer 3* interface used to measure SNMP data. If we wanted to specify the two common *directions* (*in* and *out*) we could construct a chain similar to the below example.

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m1">
  <netutil:subject xmlns:netutil="http://ggf.org/ns/nmwg/characteristic/utilization/2.0/" id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
      <nmwgt:capacity>1000000000</nmwgt:capacity>
    </nmwgt:interface>
  </netutil:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/tools/snmp/2.0</nmwg:eventType>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m2" metadataIdRef="m1">
  <netutil:subject xmlns:netutil="http://ggf.org/ns/nmwg/characteristic/utilization/2.0/" id="s2">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:direction>in</nmwgt:direction>
    </nmwgt:interface>
  </netutil:subject>
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m3" metadataIdRef="m1">
  <netutil:subject xmlns:netutil="http://ggf.org/ns/nmwg/characteristic/utilization/2.0/" id="s3">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:direction>out</nmwgt:direction>
    </nmwgt:interface>
  </netutil:subject>
</nmwg:metadata>
```

Note that the chaining is performed via the use of the *metadataIdRef* tag in the metadata element. This is a signal for services (specifically perfSONAR services such as the SNMP MA or RRD MA) to keep looking deeper in an effort to resolve the chains. The Figure 5 demonstrates the linking between the metadata elements. The resulting XML structure after chaining is also listed below.

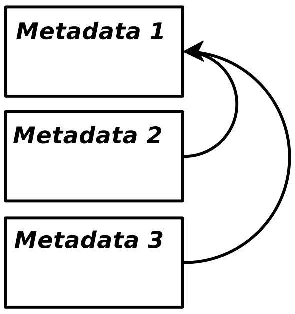

Figure 5: Graphical representation of chaining.

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m1">
  <netutil:subject xmlns:netutil="http://ggf.org/ns/nmwg/characteristic/utilization/2.0/" id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
      <nmwgt:capacity>1000000000</nmwgt:capacity>
    </nmwgt:interface>
  </netutil:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/tools/snmp/2.0</nmwg:eventType>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m2" metadataIdRef="m1">
  <netutil:subject xmlns:netutil="http://ggf.org/ns/nmwg/characteristic/utilization/2.0/" id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
      <nmwgt:capacity>1000000000</nmwgt:capacity>
      <nmwgt:direction>in</nmwgt:direction>
    </nmwgt:interface>
  </netutil:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/tools/snmp/2.0</nmwg:eventType>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m3" metadataIdRef="m1">
  <netutil:subject xmlns:netutil="http://ggf.org/ns/nmwg/characteristic/utilization/2.0/" id="s1">
```

```
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
      <nmwgt:capacity>1000000000</nmwgt:capacity>
      <nmwgt:direction>out</nmwgt:direction>
    </nmwgt:interface>
  </netutil:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/tools/snmp/2.0</nmwg:eventType>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
</nmwg:metadata>
```

For continuity, this example has not attempted to modify the *metadataIdRef* attribute. Implementations may choose to do so if they feel the need. Because eventTypes may be repeated (either as the *eventType* element or as *parameters*) we must take special care when merging them. The next example features multiple eventType merging. This example also features a so called *double chain* where the results of the first chaining operation must feed into the process for the second. This is a common occurrence, and should be supported in services.

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m1">
  <nmwg:subject id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
      <nmwgt:capacity>1000000000</nmwgt:capacity>
      <nmwgt:direction>in</nmwgt:direction>
    </nmwgt:interface>
  </nmwg:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/tools/snmp/2.0</nmwg:eventType>
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m2" metadataIdRef="m1">
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m3" metadataIdRef="m2">
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/errors/2.0</nmwg:eventType>
</nmwg:metadata>
```

The resulting output and cartoon are pictured below. We did take two major issues into consideration: multiple *parameters* and *eventType* elements that did conflict, and the double chaining. Services that do not support multiple eventTypes (or simply wish to not implement a naive form of chaining) should not worry about special cases such as Figure 6.

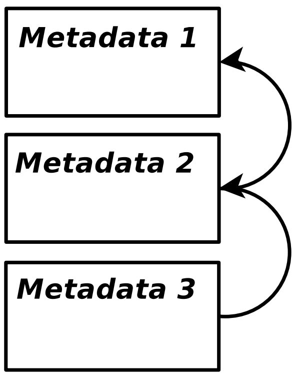

Figure 6: Alternate graphical representation of chaining.

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m1">
  <nmwg:subject id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
      <nmwgt:capacity>1000000000</nmwgt:capacity>
      <nmwgt:direction>in</nmwgt:direction>
    </nmwgt:interface>
  </nmwg:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/tools/snmp/2.0</nmwg:eventType>
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m2" metadataIdRef="m1">
  <nmwg:subject id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
      <nmwgt:capacity>1000000000</nmwgt:capacity>
      <nmwgt:direction>in</nmwgt:direction>
    </nmwgt:interface>
  </nmwg:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/tools/snmp/2.0</nmwg:eventType>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m3" metadataIdRef="m2">
  <nmwg:subject id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
      <nmwgt:capacity>1000000000</nmwgt:capacity>
      <nmwgt:direction>in</nmwgt:direction>
    </nmwgt:interface>
  </nmwg:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/tools/snmp/2.0</nmwg:eventType>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/errors/2.0</nmwg:eventType>
</nmwg:metadata>
```

Services may treat particular elements (such as eventTypes and parameters with certain *name* attributes) in a special way. The service is careful not to overwrite or lose any information and will only *add* these items together. This is not the case for any element though, consider the following example.

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m1">
  <nmwg:subject id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
      <nmwgt:capacity>1000000000</nmwgt:capacity>
      <nmwgt:direction>in</nmwgt:direction>
    </nmwgt:interface>
  </nmwg:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/tools/snmp/2.0</nmwg:eventType>
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m2" metadataIdRef="m1">
  <nmwg:subject id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifName>eth1</nmwgt:ifName>
      <nmwgt:direction>out</nmwgt:direction>
    </nmwgt:interface>
  </nmwg:subject>
</nmwg:metadata>
```

Note that we probably wanted to change the direction for this particular interface, not necessarily the *ifName* element. The output of this chain is shown below.

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m1">
```

```
  <nmwg:subject id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
      <nmwgt:capacity>1000000000</nmwgt:capacity>
      <nmwgt:direction>in</nmwgt:direction>
    </nmwgt:interface>
  </nmwg:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/tools/snmp/2.0</nmwg:eventType>
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m2" metadataIdRef="m1">
  <nmwg:subject id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth1</nmwgt:ifName>
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
      <nmwgt:capacity>1000000000</nmwgt:capacity>
      <nmwgt:direction>out</nmwgt:direction>
    </nmwgt:interface>
  </nmwg:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/tools/snmp/2.0</nmwg:eventType>
</nmwg:metadata>
```

This example shows that it is very easy to introduce semantic errors when designing a chain instance. It also shows that the service may not be interested in protecting a poorly designed chain from being accepted. It is possible to build in different rules instead of *last seen value* such as *first seen*, *original*, or other combinations. It is imperative that services describe their own implementations of chaining, particularly when interoperability becomes an issue.

A final example comes when we deal with items with the same *localname*, but perhaps a different *namespace*. There are several approaches that can be taken to dealing with this type of situation. The SNMP example follows a safe approach of simply adding all of the elements in question and not attempting to internally merge at all. This causes *unreadable* metadata in many cases, but does not permit *data pollution*.

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m1">
  <nmwg:subject id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
    </nmwgt:interface>
  </nmwg:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m2" metadataIdRef="1">
  <netutil:subject xmlns:netutil="http://ggf.org/ns/nmwg/characteristic/utilization/2.0/" id="s2">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
      <nmwgt:direction>in</nmwgt:direction>
      <nmwgt:capacity>1000000000</nmwgt:capacity>
    </nmwgt:interface>
  </netutil:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
</nmwg:metadata>
```

There are three approaches that I will illustrate here: *safe yet stupid*, *dangerous yet intelligent*, and finally *slow and steady*. The last approach is sometimes used in practice; finding the proper balance will require some thought (depending on how sensing or accurate a service wishes to become. Approach one yields output similar to the below example.

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m1">
  <nmwg:subject id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
    </nmwgt:interface>
  </nmwg:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m2" metadataIdRef="1">
  <netutil:subject xmlns:netutil="http://ggf.org/ns/nmwg/characteristic/utilization/2.0/" id="s2">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
      <nmwgt:direction>in</nmwgt:direction>
      <nmwgt:capacity>1000000000</nmwgt:capacity>
    </nmwgt:interface>
  </netutil:subject>
  <nmwg:subject id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
    </nmwgt:interface>
  </nmwg:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
</nmwg:metadata>
```

Note that this is not schema valid, and presumably would not return results from the backend storage. This is rather ironic given that we are trying to preserve validity on the schema side, yet still generate a clearly invalid result. The other end of the spectrum gives a result such as the example below.

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m1">
  <nmwg:subject id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
    </nmwgt:interface>
  </nmwg:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m2" metadataIdRef="1">
  <netutil:subject xmlns:netutil="http://ggf.org/ns/nmwg/characteristic/utilization/2.0/" id="s2">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
      <nmwgt:direction>in</nmwgt:direction>
      <nmwgt:capacity>1000000000</nmwgt:capacity>
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
    </nmwgt:interface>
  </netutil:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
</nmwg:metadata>
```

The *stupid* part of this comes from not caring about *namespaces*, and only merging based on *localname*. Because the source metadata featured the *netutil* namespace it remains and all other items are added to it.

The approach taken by some services is to have a little *domain* knowledge before making a quick judgement. Knowing full well that *nmwg* is a more general namespace than *netutil*, the service tries to guess the intent and goes with the most general namespace in order to support a richer query set. Internally anything that utilizes the *nmwg* namespace receives a wild card when performing searches. When we are faced with a choice between specific and general, the service errs on the side of general. An example of this merge is below.

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m1">
  <nmwg:subject id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
    </nmwgt:interface>
  </nmwg:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
```

```
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m2" metadataIdRef="1">
  <nmwg:subject id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
      <nmwgt:direction>in</nmwgt:direction>
      <nmwgt:capacity>1000000000</nmwgt:capacity>
    </nmwgt:interface>
  </nmwg:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
</nmwg:metadata>
```

A final question remains: what happens if you are dealing with two very specific namespaces such as this example.

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m1">
  <neterr:subject xmlns:neterr="http://ggf.org/ns/nmwg/characteristic/errors/2.0/" id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
    </nmwgt:interface>
  </neterr:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/errors/2.0</nmwg:eventType>
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m2" metadataIdRef="1">
  <netutil:subject xmlns:netutil="http://ggf.org/ns/nmwg/characteristic/utilization/2.0/" id="s2">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:hostName>localhost</nmwgt:hostName>
    </nmwgt:interface>
  </netutil:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
</nmwg:metadata>
```

Some services will still guess “general” and convert to the *nmwg* namespace. The resulting data set will take on an interesting look:

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m1">
  <neterr:subject xmlns:neterr="http://ggf.org/ns/nmwg/characteristic/errors/2.0/" id="s1">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
    </nmwgt:interface>
  </neterr:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/errors/2.0</nmwg:eventType>
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m2" metadataIdRef="1">
  <nmwg:subject id="s2">
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
    </nmwgt:interface>
  </nmwg:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/errors/2.0</nmwg:eventType>
</nmwg:metadata>
```

Clearly the two eventTypes (for utilization and errors) may not appear in the same metadata description, but again the service can try to help out a bit. eventType descriptions are interpreted as *or* operations when performing a query. Therefore even if our chain was constructed poorly, our final results will be rather robust (perhaps a bit more robust than needed). The service designers will no doubt settle on an approach that fits well for the data they are exposing.

5.2 Filter Chaining

Filter chaining involves the application of a *filter* (or function) to the underlying dataset that a particular metadata describes. We can think of this much like a database operation, where the first metadata is used to select a broad range of data, and subsequent metadata elements that are chained in this manner are used to slowly whittle down the dataset to a very specific range.

Figure 7 illustrates the distinction between the various operators of a filter chain. The circles themselves represent the actual metadata description of a dataset (taken from the universe of all data). The intersection of these two metadata descriptions becomes the data set that we are interested in.

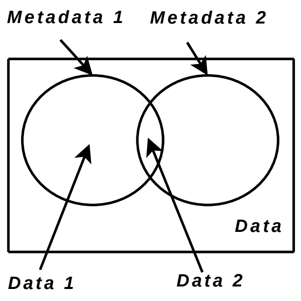

Figure 7: Diagram showing the intersection of information sets.

It is important to note that even though we are manipulating the data through this form of chaining, we should not be harming it, or the related metadata elements. Chaining in general is a non-destructive operation, although it is very possible that when implemented poorly response data corruption may occur.

Filter operations themselves can vary from time range selection to aggregations such as performing a cumulative distribution function (CDF). Describing all possible operators is well beyond the scope of this work. Current experience has named most statistical and database operations as candidates for filtering, although new uses being devised.

5.2.1 Operator Chaining Examples

Filter chaining is an easier concept to manage than merge chaining, partially because there are less rules and nuances to grasp. As stated above, it is easy to think of the dataset for the source metadata to be *input* to a function that is named by the metadata utilizing the filter chain. Consider Figure 8 as an example of the internal process of resolving a filter chain.

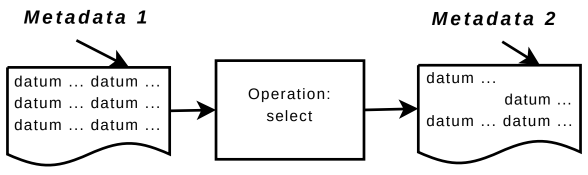

Figure 8: Graphical results of a filtering step on a dataset.

The syntax of filter chaining is similar to that of merge chaining (by using *metadataIdRef* attributes) but the placement is a bit different. Consider this example.

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m1">
  <netutil:subject xmlns:netutil="http://ggf.org/ns/nmwg/characteristic/utilization/2.0/" id="s1">
```

```
    <nmwgt:interface xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
      <nmwgt:ifAddress type="ipv4">127.0.0.1</nmwgt:ifAddress>
      <nmwgt:hostName>localhost</nmwgt:hostName>
      <nmwgt:ifName>eth0</nmwgt:ifName>
      <nmwgt:ifIndex>2</nmwgt:ifIndex>
      <nmwgt:direction>in</nmwgt:direction>
      <nmwgt:capacity>1000000000</nmwgt:capacity>
    </nmwgt:interface>
  </netutil:subject>
  <nmwg:eventType>http://ggf.org/ns/nmwg/tools/snmp/2.0</nmwg:eventType>
  <nmwg:eventType>http://ggf.org/ns/nmwg/characteristic/utilization/2.0</nmwg:eventType>
</nmwg:metadata>

<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="m2">
  <select:subject id="s2" metadataIdRef="m1" xmlns:select="http://ggf.org/ns/nmwg/ops/select/2.0/"/>
  <select:parameters id="param2c" xmlns:select="http://ggf.org/ns/nmwg/ops/select/2.0/">
    <nmwg:parameter name="startTime">1121472000</nmwg:parameter>
    <nmwg:parameter name="endTime">1121904000</nmwg:parameter>
    <nmwg:parameter name="consolidationFunction">AVERAGE</nmwg:parameter>
    <nmwg:parameter name="resolution">60</nmwg:parameter>
  </select:parameters>
  <nmwg:eventType>http://ggf.org/ns/nmwg/ops/select/2.0</nmwg:eventType>
</nmwg:metadata>
```

The reference is placed in the *subject* element in this case, as in merge chaining this is a signal to the service that filter chaining may be required. This indicates that the *input* is the data pointed to by the first metadata and the *output* will be a subset of this. For the sake of these examples we will be dealing with the *select* namespace as our filter of choice due to an abundance of examples and its common goal of filtering based on time. Other filter examples should work in the same manner.

Because the operations of a filter chain are essentially *internal* we do not present what resultant XML should look like. Currently services ignore many of the steps that may go into reforming the XML for response messages in favor of simply returning the *backend* representation of metadata. While quick and easy, this does lead to information loss (specifically when dealing with the various ways to implement merge chaining). Client applications may have no reason to see the original filter information, and therefore are built not to need it.

6 Result Codes

Result and status codes are an important part of an interactive service; these values are used to to convey information about the system during operation. Client and server software can consume these results, perform a simple lookup, and return useful information back to users. This section will describe:

- A hierarchy of result codes based loosely on similar efforts in other protocols
- Guidelines for the use of these codes within software
- Guidelines for the types of textual messages that will accompany these codes

6.1 Syntax

A **STATUS_CODE** is defined by the following pattern:

```
STATUS_CODE =
                    STATUS_PREFIX "/" STATUS_CATEGORY "/" STATUS_NAME "/" VERSION "/"

STATUS_CATEGORY =
                    "informational"
                    | "successful"
                    | "redirection"
                    | "clienterror"
                    | "servererror"
```

STATUS_PREFIX is the start of the URI, and will be defined as "http//schemas.ogf.org/nmc/status". The **VERSION** is defined to be a string presenting information about the version of protocol, e.g. *201109* or *20110925*. This version is suggested to remain "date" based instead of a haphazard assignment of numbers; the former will impart additional information about when a particular version entered use. Lastly, note that the final "/" is required in this format.

The following sections present acceptable status names for certain category.

6.1.1 Informational

```
STATUS_NAME =
                "protocol_version"
                | "data_limitation"
                | "service_contact"
```

6.1.2 Successful

```
STATUS_NAME =
                NULL
```

This status is left intentionally blank.

6.1.3 Redirection

```
STATUS_NAME =
                NULL
```

This status is left intentionally blank.

6.1.4 Clienterror

```
STATUS_NAME =
                    "bad_message"
                    | "bad_request"
                    | "authentication_failed"
                    | "unauthorized"
                    | "message_not_allowed"
                    | "event_type_not_allowed"
                    | "request_too_large"
                    | "request_timeout"
                    | "protocol_not_allowed"
                    | "chaining_not_understood"
```

6.1.5 Servererror

```
STATUS_NAME =
                    "data_fetch_error"
                    | "too_busy"
                    | "administrative_down"
```

6.2 Semantics

The following categories were chosen to classify errors. Note that these recommendations form the basis of the standard, extension by implementations may be more specific as required. E.g. we anticipate that an implementation may choose to offer a specific category that further extends something in this spec. For example “http://schemas.ogf.org/nmc/status/servererror/data_fetch_error/database_down/201109/” may be an extension of “http://schemas.ogf.org/nmc/status/servererror/data_fetch_error/201109/”. This does not go against the spirit of this specification. Client applications should be advised that parsing an error code may result in seeing this “unexpected” last part, and could terminate parsing up to this point to avoid a complete failure.

A final note relates to the location of the “version” string, e.g. the date the status was last modified by NMC. Conventional wisdom notes that namespaces should be constructed with a date in the middle of the string [4]. Experience has found that this date is more effective at the end, and allows for updates to certain parts of the schematic standard, without re-writing all of them.

6.2.1 Informational

This represents valid responses for informational requests. Using just the top level, e.g. "http://schemas.ogf.org/nmc/status/informational/201109/" is considered to be acceptable. The following subclasses were identified:

- **protocol_version**: Returns the version of the NMC protocol in use
- **data_limitation**: Returns a message indicating that responses will be limited to a pre-set range or size
- **service_contact**: Returns the contact information (e.g. administrative contacts, etc.) for the service

6.2.2 Successful

This represents valid responses for any form of successful interaction. Using just the top level, e.g. "http://schemas.ogf.org/nmc/status/successful/201109/" is considered to be the only acceptable response.

6.2.3 Redirection

This represents valid responses for any form of redirection that the service deems acceptable. Using just the top level, e.g. "http://schemas.ogf.org/nmc/status/redirection/201109/" is considered to be the only acceptable response. This redirection activity is assumed to be "temporary", e.g. clients should not cache/store this redirection for any reason.

6.2.4 Clienterror

This represents an error issued to a client based on the request. Use of the top level, e.g. "http://schemas.ogf.org/nmc/status/clienterror/201109/" may be possible, but is not recommended. The following subclasses were identified:

- **bad_message**: Returns a message indicating there is a syntactic (XML based) or semantic (logical structure of request) error. Context will be given in human readable text.
- **bad_request**: Request was send to non-existent endpoint on the node/service in question
- **authentication_failed**: The service could not determine who the user really was
- **unauthorized**: The user is not allowed to request the content/resource
- **message_not_allowed**: The wrong type of message was sent to the service (indicates a deeper level of semantic checking beyond **bad_message**)
- **event_type_not_allowed**: The eventType is not allowed or unsupported by this service (indicates a deeper level of semantic checking beyond **bad_message**)
- **request_too_large**: The request message was too large to process
- **request_timeout**: The request has taken too long to service

- **protocol_not_allowed**: Version of NMC protocol was not understood between the client and server. We choose to draw the line at NMC protocol in this case, and not "higher" into the Network Measurement description, or "lower" into SOAP, HTTP, TCP, etc. In the event that we cannot handle schematic nuances beyond NMC (or lower layer issues from something else) we should fall back to the general error, or **bad_message**
- **chaining_not_understood**: The chaining used in the message, either merge or operation, was not understood by the parser. This is a specific use case of **bad_message**

6.2.5 Servererror

This represents an error issued to a client based on the behavior of the service that s serving the request. Using just the top level, e.g. "http://schemas.ogf.org/nmc/status/servererror/201109/" is considered to be acceptable, but not recommended. The following subclasses were identified:

- **data_fetch_error**: The request is valid, but there is an underlying problem with the service backend.
- **too_busy**: The service is unable to act on the request at this time due to internal limitations on resource consumption
- **administrative_down**: The service has been configured to not respond.

6.3 Use Cases

There are two primary use cases of these coded values:

- **Basic Structure**: Using the codes and user meanings as prescribed in this document
- **Extension**: Adding new codes, or alternate meanings, to extend this document

6.3.1 Basic Structure

Two items which characterize the basic massage structure containing a status code are as follows:

- The namespace "http://ggf.org/ns/nmwg/result/2.0/" of xml tags containing status code information.
- The "metadata" containing **STATUS_CODE** and "data" with **SHORT_DESCRIPTION** of a status code.

The following xml snippet presents the structure:

```
<nmwg:message xmlns:nmwg=http://ggf.org/ns/nmwg/base/2.0/
              xmlns:nmwgr="http://ggf.org/ns/nmwg/result/2.0/">

  <nmwg:metadata id="status-code">
    <nmwg:eventType>
      STATUS_CODE
    </nmwg:eventType>
  </nmwg:metadata>

  <nmwg:data id="status-code-desc"
             metadataIdRef="status-code">
    <nmwgr:datum>
      SHORT_DESCRIPTION
    </nmwgr:datum>
  </nmwg:data>

</nmwg:message>
```

6.3.2 Extension

The basic structure of status code message can be extended by introducing new namespaces (see **new-namespace** in the xml snippet below). They allow to redefine the datum element in order to contain more complex information formats.

```
<nmwg:message xmlns:nmwg=http://ggf.org/ns/nmwg/base/2.0/>

  <nmwg:metadata id="status-code">
    <nmwg:eventType>
      STATUS_CODE
    </nmwg:eventType>
  </nmwg:metadata>

  <nmwg:data id="status-code-desc"
             metadataIdRef="status-code">
    <new-namespace:datum>
      STATUS_EXTENDED_CONTENT
    </new-namespace:datum>
  </nmwg:data>

</nmwg:message>
```

6.4 Examples

The following examples show use of the result codes.

6.4.1 Successful Code

```
<nmwg:message id="response" type="EchoResponse"
              xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
              xmlns:nmwgr="http://ggf.org/ns/nmwg/result/2.0/">

  <nmwg:metadata id="status-code">
    <nmwg:eventType>
      http://schemas.ogf.org/nmc/status/successful/201109/
    </nmwg:eventType>
  </nmwg:metadata>

  <nmwg:data id="status-code-desc"
             metadataIdRef="status-code">
    <nmwgr:datum>
      This is the success echo response from the service.
    </nmwgr:datum>
  </nmwg:data>

</nmwg:message>
```

6.4.2 Servererror Code

```
<nmwg:message id="resp1"
              type="MetadataKeyResponse"
              xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
              xmlns:nmwgr="http://ggf.org/ns/nmwg/result/2.0/">

  <nmwg:metadata id="status-code">
    <nmwg:eventType>
      http://schemas.ogf.org/nmc/status/servererror/data_fetch_error/201109/
    </nmwg:eventType>
  </nmwg:metadata>

  <nmwg:data id="status-code-desc"
             metadataIdRef="status-code">
    <nmwgr:datum>
      Requested metadata items are not available.
    </nmwgr:datum>
  </nmwg:data>

</nmwg:message>
```

7 Extension

This section must become the basis for all extension protocols. As a demonstration we include a protocol that must be implemented by all measurement services: the *Echo Protocol*. This protocol will incorporate the preceding work to eliminate duplication as much as possible, only specifying parts that are necessary for clarification. Each protocol extension may be treated as a separate work, and will include the necessary schema, analysis, and example sections.

7.1 Echo Protocol

The sole purpose of certain services, in a measurement framework, is to aid in the discovery and protection of the enterprise. The tasks undertaken by these critical components also require sound communication protocols based on the same formats used to exchange and store measurement data as defined by the **NM-WG** [8].

The *Echo Protocol* can be used by client applications as well as other services to ascertain the *liveness* of a given service. A well formatted **EchoRequest** message, when sent to a service, should trigger a similar **EchoResponse**. This interaction allows a client or service to gauge the responsiveness of a service; the potential to learn more information is also available for services who wish to implement more functionality.

The core functionality of the *Echo Protocol* is to provide a simple *request* and *response* capable of delivering rudimentary status information. This protocol for exchange is similar to other types of communication, notably *ping*. While this protocol may seem to be a reinvention of existing tooling, the extension possibility far outweighs the duplication of functionality.

We present an overview of the messages used in this protocol, including both schematic designs and examples for the *Request Message* (see Section 7.1.2) and *Response Message* (see Section 7.1.3). We conclude with a brief description of where extensions are possible followed by some current examples in *Protocol Extension* (see Section 7.1.5).

7.1.1 Architecture

To ensure availability, each service must be able to respond to simple queries regarding status. Services that fail to answer a direct question may be experiencing difficulty, and therefore may not be able to complete interaction with interested parties. Client applications, services, or external monitoring tools should use this simple method to quickly come to conclusions regarding framework availability.

All services must contain the ability to respond to the most basic of *Echo Protocol* messages as described by this document. The minimum requirement of an *Echo Protocol* exchange is simply responding to a properly encoded request. *Echo Protocol* extensions may be built from this general protocol to elicit additional functionality on a service by service basis to do tasks such as test the capabilities of the service, receive statistics, or monitor erroneous behavior. The assignment of these other tasks within an **EchoRequest** message is valid provided that the basic structure is not compromised.

7.1.2 Request Message

The **EchoRequest** message can be initiated by a client application or service wanting to know the availability of some other service. The format of this message is minimal with respect to other protocol messages as the input is rather simple. The basic format described in this work for measurements has been adapted as a template for use in service communication as well, keeping the concept of metadata and data intact.

7.1.2.1 Request Message Schema

The following schema is a native description of the request schema as in the RELAX-NG[9] language. Through the use of tools such as Trang[11] and MSV[5] it is possible to convert this to other widely accepted formats such as XSD[12]. The following describes the **EchoRequest** schema. Note that this will only validate **EchoRequest** messages.

```
# Begin Schema

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

start =
  element nmwg:message {
    attribute id { xsd:string } &
    attribute messageIdRef { xsd:string }? &
    attribute type {
      "EchoRequest" |
      "http://schemas.perfsonar.net/messages/EchoRequest/1.0"
    } &
    element nmwg:metadata {
      attribute id { xsd:string } &
      element nmwg:eventType {
        "http://schemas.perfsonar.net/tools/admin/echo/2.0"
      }
    } &
    element nmwg:data {
      attribute id { xsd:string } &
      attribute metadataIdRef { xsd:string }
     }
   }

# End Schema
```

7.1.2.2 Request Message Analysis

The following is a breakdown of the elements featured in the schema.

7.1.2.2.1 Message

```
<nmwg:message  xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
               type="EchoRequest"
               id="STRING">

  <nmwg:metadata />

  <nmwg:data />

</nmwg:message>
```

Table 16: Message Element Specifics

Message Element	
localname	message
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id, type
nested elements	metadata, data
required	yes

This appears the same was as it does in Section 4.3.2.1, the only notable exception is a requirement that the *type* attribute contain the values **EchoRequest** or **http://schemas.perfsonar.net/messages/EchoRequest/1.0**.

7.1.2.2.2 Metadata

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="STRING">

   <nmwg:eventType />

</nmwg:metadata>
```

Table 17: Metadata Element Specifics

Metadata Element	
localname	metadata
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id, MetadataIdRef
nested elements	eventType
required	yes

This appears the same was as it does in Section 4.3.2.4, the only exception is specifying that *EventType* can be the *only* child.

7.1.2.2.3 EventType

```
<nmwg:eventType xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/">
  http://schemas.perfsonar.net/tools/admin/echo/2.0
</nmwg:eventType>
```

Table 18: EventType Element Specifics

EventType Element	
localname	eventType
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	N/A
nested elements	text
required	yes

The *eventType* element is normally used to specify an action for a service or measurement. We utilize it for this role in the *Echo Protocol* by specifying the action of responding to an **EchoRequest**. There are no attributes permitted for this element, and only text can be used as a child, specifically text reporting *http://schemas.perfsonar.net/tools/admin/echo/2.0*.

Because this element is currently well defined into a specific role and purpose, the eventType is non-negotiable. **Extensions**, as discussed in Section 7.1.5, may be employed on a service by service basis to expand this basic specification, as long as the role is preserved.

7.1.2.2.4 Data

```
<nmwg:data xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
           id="STRING"
           metadataIdRef="STRING" />
```

Table 19: Data Element Specifics

Data Element	
localname	data
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id, metadataIdRef
nested elements	N/A
required	yes

This appears the same was as it does in Section 4.3.2.8.

7.1.2.3 Request Message Example

The first example shows a correct configuration for an **EchoRequest** message.

```
<!-- Begin XML -->

<message xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
         type="EchoRequest"
         id="message1">

  <metadata id="m1">
    <nmwg:eventType>
      http://schemas.perfsonar.net/tools/admin/echo/2.0
    </nmwg:eventType>
  </metadata>

  <data id="d1" metadataIdRef="m1" />

</message>

<!-- End XML -->
```

The final example shows two incorrect items: the message *type* and *eventType* are both wrong. This must be rejected by a service.

```
<!-- Begin XML -->

<message xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
         type="Echo"
         id="message2">

  <metadata id="m1">
    <nmwg:eventType>
      echo
    </nmwg:eventType>
  </metadata>

  <data id="d1" metadataIdRef="m1" />

</message>

<!-- End XML -->
```

7.1.3 Response Message

The **EchoResponse** message is a reply to a given **EchoRequest** message from a client application or service. The format of this message is minimal with respect to other protocol messages as the input is rather simple. The basic format described in this work for measurements has been adapted as a template for use in service communication as well, keeping the concept of metadata and data intact.

7.1.3.1 Response Message Schema

The following schema is a native description of the response schema as in the RELAX-NG[9] language. Through the use of tools such as Trang[11] and MSV[5] it is possible to convert this to other widely accepted formats such as XSD[12]. The following describes the **EchoResponse** schema. Note that this will only validate **EchoResponse** messages.

```
# Begin Schema

namespace nmwg="http://ggf.org/ns/nmwg/base/2.0/"
namespace nmwgr="http://ggf.org/ns/nmwg/result/2.0/"

start =
  element nmwg:message {
    attribute id { xsd:string } &
    attribute messageIdRef { xsd:string }? &
    attribute type {
      "EchoResponse" |
      "http://schemas.perfsonar.net/messages/EchoResponse/1.0"
    } &
    element nmwg:metadata {
      attribute id { xsd:string } &
      element nmwg:eventType {
        xsd:string
      }
    } &
    element nmwg:data {
      attribute id { xsd:string } &
      attribute metadataIdRef { xsd:string } &
      element nmwgr:datum {
        xsd:string |
        attribute value { xsd:string }
      } |
      element nmwg:datum {
        xsd:string |
        attribute value { xsd:string }
      } |
    }
  }

# End Schema
```

7.1.3.2 Response Message Analysis

The following is a breakdown of the elements featured in the schema.

7.1.3.2.1 Message

```
<nmwg:message  xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
             type="EchoResponse"
             id="STRING">

  <nmwg:metadata />

  <nmwg:data />

</nmwg:message>
```

This appears the same was as it does in Section 4.4.2.1, the only exception is a requirement that the *type* attribute contain the values **EchoResponse** or **http://schemas.perfsonar.net/messages/EchoResponse/1.0**.

7.1.3.2.2 Metadata

```
<nmwg:metadata xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/" id="STRING">

   <nmwg:eventType />

</nmwg:metadata>
```

Table 20: Message Element Specifics

Message Element	
localname	message
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id, type
nested elements	metadata, data
required	yes

Table 21: Metadata Element Specifics

Metadata Element	
localname	metadata
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id, metadataIdRef
nested elements	eventType
required	yes

This appears the same was as it does in Section 4.4.2.4, the only exception is specifying that *EventType* can be the *only* child.

7.1.3.2.3 EventType

```
<nmwg:eventType xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/">
  http://schemas.perfsonar.net/tools/admin/echo/2.0
</nmwg:eventType>

or

<nmwg:eventType xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/">
  http://schemas.perfsonar.net/status/success/echo/1.0
</nmwg:eventType>
```

Table 22: EventType Element Specifics

EventType Element	
localname	eventType
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	N/A
nested elements	text
required	yes

The *eventType* element is normally used to specify an *action* for a service or measurement. We utilize it for this role in the *Echo Protocol* by specifying the action of a response to an **EchoRequest**. There are no attributes permitted for this element, and only text can be used as a child, specifically text reporting the *status* of the transaction. A complete list of available status strings is available in Section 7.1.4.

7.1.3.2.4 Data

```
<nmwg:data xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
           id="STRING"
           metadataIdRef="STRING" />
```

Table 23: Data Element Specifics

Data Element	
localname	data
namespaces	http://ggf.org/ns/nmwg/base/2.0/
attributes	id, metadataIdRef
nested elements	datum
required	yes

This appears the same was as it does in Section 4.4.2.5 with the exception of allowing **Datum** as a child element.

7.1.3.2.5 Datum

```
<nmwgr:datum xmlns:nmwgr="http://ggf.org/ns/nmwg/result/2.0">
  TEXT
</nmwgr:datum>

<!-- OR -->

<nmwg:datum xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/">
  TEXT
</nmwg:datum>
```

Table 24: Datum Element Specifics

Datum Element	
localname	datum
namespaces	http://ggf.org/ns/nmwg/result/2.0/, http://ggf.org/ns/nmwg/base/2.0/
attributes	value
nested elements	text
required	yes

The *datum* element describes measurements in most circumstances; the intent in the *Echo Protocol* is to report back a human readable *status* message. There is only one possible attribute accepted for this element, *value*, and it may be used in place of an enclosed text element. The text could be any message the service wishes to return.

7.1.3.3 Response Message Example

The first example shows a correct configuration for an **EchoResponse** message.

```
<!-- Begin XML -->
```

```
<nmwg:message xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
              xmlns:nmwgr="http://ggf.org/ns/nmwg/result/2.0/"
              type="EchoResponse"
              id="message3"
              messageIdRef="message1">

  <metadata id="m1">
    <nmwg:eventType>success.echo</nmwg:eventType>
  </metadata>

  <data id="d1" metadataIdRef="m1">
    <nmwgr:datum>The echo request has passed.</nmwgr:datum>
  </data>

</nmwg:message>

<!-- End XML -->
```

The final example shows the result of a failed **EchoRequest**.

```
<!-- Begin XML -->

<nmwg:message xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
              xmlns:nmwgr="http://ggf.org/ns/nmwg/result/2.0/"
              type="EchoResponse"
              id="message4"
              messageIdRef="message2">

  <metadata id="m1">
    <nmwg:eventType>error.echo</nmwg:eventType>
  </metadata>

  <data id="d1" metadataIdRef="m1">
    <nmwgr:datum>The echo request has failed.</nmwgr:datum>
  </data>

</nmwg:message>

<!-- End XML -->
```

7.1.4 Result Codes

The following new result codes can be incorporated into the echo protocol based on Section 6. We will introduce these into both styles to allow for backwards compatibility. The original style is presented first:

```
success.
        echo
error.
        echo
```

We can express the same information using the new URI style:

```
http://schemas.perfsonar.net/status/
                              success/
                                      echo/
                                            1.0
                              error/
                                      echo/
                                            1.0
```

Note that it is possible to add more specific error conditions as the functionality of this protocol increases. For example, if we are testing database connectivity it may make sense to add an *echo.db.* range of status. This is left as an exercise for extensions to this protocol.

7.1.5 Protocol Extension

There are two avenues for extension within the *Echo Protocol* as described in this document. It is possible to manipulate the values contained within the *eventType* to advance functionality, or through *schema modification* it is possible to add additional elements capable of handling a wider range of actions. Extensions that modify the schema for a given service must not change the themes presented in this protocol specification. It is imperative that all services respect the basic functionality in their quest to add new features.

7.1.5.1 eventType Extension

The current accepted eventType for the *Echo Protocol*'s **EchoRequest** message is **http://schemas.perfsonar.net/tools/admin/echo/2.0**. This *action* must be accepted by all services. By adding additional eventTypes with the same format it is possible to extract additional information via a service.

Consider simple service X. The designer of this service wishes to create a special behaviour for specific eventTypes. The following new eventTypes are added to her service code (and to her implementation of the schema):

- **http://schemas.perfsonar.net/tools/admin/echo/X/2.0 -** Allows service contact information to be returned via *nmwgr:datum*
- **http://schemas.perfsonar.net/tools/admin/echo/X/contact/2.0 -** Allows service contact information to be returned via *nmwgr:datum*
- **http://schemas.perfsonar.net/tools/admin/echo/X/stats/2.0 -** Allows service usage statistics to be returned via *nmwgr:datum*
- **http://schemas.perfsonar.net/tools/admin/echo/X/db/2.0 -** Allows a basic database test to be performed, the results of which are returned via *nmwgr:datum*

By simply allow some additional string matching to occur in the eventType it is now possible to receive additional data to check the health and status of the system.

7.1.5.2 Other Extensions

Similar to the above approach, it is possible to extend the schema by adding additional elements to increase functionality. Individuals pursuing this route must be comfortable with schema design in general and the layout of the **NM-WG** and **NMC-WG** schema descriptions specifically.

A simple extension involves allowing the commonly used parameters structure to reside in the **Message** of the **EchoRequest** message. This modification is presented below.

```
# Begin Schema

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

start =
  element nmwg:message {
    attribute id { xsd:string } &
    attribute messageIdRef { xsd:string }? &
    attribute type {
      "EchoRequest" |
      "http://schemas.perfsonar.net/messages/EchoRequest/1.0"
    } &
    element nmwg:metadata {
      attribute id { xsd:string } &
      element nmwg:eventType {
        "http://schemas.perfsonar.net/tools/admin/echo/2.0"
      } &
      element nmwg:parameters {
        element nmwg:parameter {
          attribute name { xsd:string }
        }
      }?
    } &
    element nmwg:data {
      attribute id { xsd:string } &
      attribute metadataIdRef { xsd:string }
    }
  }

# End Schema
```

Building on the example in Section 7.1.5.1, the following example message shows how to ask for similar information as previously described.

```
<nmwg:message type="http://schemas.perfsonar.net/messages/EchoRequest/1.0"
              id="message.96587"
              xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/">

  <nmwg:metadata id="metadata.21324">
    <nmwg:eventType>
    http://schemas.perfsonar.net/tools/admin/echo/2.0
    </nmwg:eventType>
    <nmwg:parameters>
      <nmwg:parameter name="status" />
    </nmwg:parameters>
  </nmwg:metadata>

  <nmwg:data id="data.54365" metadataIdRef="metadata.21324"/>

</nmwg:message>
```

While this method does require some additional schema modification, the result produced is the same as described in Section 7.1.5.1. An important consideration is the inclusion of *parameters* in an **EchoRequest**.

It must be noted that the extension methods proposed here preserve the underlying base protocol completely. Existing services that provide strict validation may reject messages that do not fit this standard explicitly, so be sure to design client applications appropriately.

8 Security Considerations

8.1 Quality of service and resource control

Any service that processes requests from external sources, will not respond or progress requests (in a timely matter) if the resources that the service acts upon are saturated. Given any request always causes some resource utilization, resources can always be saturated by simply sending a lot of requests. This is known as a Denial of Service(DoS) attack. NMC-based services might be more likely to suffer successful attacks as their underlying function(collect, perform or analyse network measurements) are resource intensive, thus requiring fewer requests to to saturate the service.

Implementers and deployers should assert how prone there service is to attack and implement, configure or deploy countermeasures. Because information on this subject matter is widely available, we will only mention that NMC facilitates Authentication and Authorization; which could play a role in protecting the service.

Beyond this, a NMC service might in turn act upon external resources. Because of this an NMC service could be prone to be abused to perform, boost or amplify DoS attacks. Related; there is the subtle problem that emerges when the external resource has some for of protection against abuse:

A NMC service may obtain a ’bad reputation’ given it emits certain request patterns, a ’bad reputation’ skews measurements. For example, given a NMC service(A) performs a measurement of resource(B), if such a measurement is performed frequent enough, resource(B) might presume that it’s being attacked by service(A) and therefore, opt to block service(A). Given that service(A) is now being blocked, the service might falsely report that resource(B) is unavailable.

These issues could be solved by tracking the actions performed by the service and based upon this information opt to not perform certain requests at that point in time. Given that this is the case opt to return to the client that it should try later(insert footnote with link to respond code) or ’throttle’ the measurements. Alternatively, if the service can assert that the action to be triggered would have a too great of an impact upon the network it can opt to refuse to perform it all together and instead inform the client that it should break the action up into smaller actions if possible.(insert footnote with link to result code)

8.2 Data protection and privacy

As outlined by Martin Swany in An Extensible Schema for Network Measurement and Performance Data[10], data passing through NMC services might be sensitive. Implementers should provide means to control distribution of such data and deployers should configure their services as well as manage the environment to prevent unauthorised access to the data.

In select cases part of the data that is invariant towards analysis can be anonymised to prevent information being available to unauthorised parties while still allowing measurements and analysis to take place. In such cases, to ensure that the information is correctly interpreted, results must always clearly advertise if and what information is anonymised.(insert link to section that describes a standardised? way to do advertise this.)

Given sensitive information is provided by a service to a authorized party one has to prevent ease-dropping(or snooping), NMC-WG protocols will provide no protection as we delegate this responsibility to the underlying transport. This means implementers and deployers should take care in choosing which transport(NMC-WG binding) to use.

9 Conclusion

The preceding work has described a simple protocol that will form the basis of communication for software exchanging both network measurements and topological information. This protocol is both minimal and flexible — extension to specific use cases is possible and expected. This work has been careful to retain the concepts described in other working groups including **NM-WG** and **NML-WG** in an effort to remain compatible with the primary data types.

10 Acknowledgements

The authors gratefully acknowledge the contributions of the perfSONAR consortium to this work. Specifically staff and member institutions from ESnet, GÉANT, Internet2, and RNP have provided extensive input and feedback into the implementation of this and all NMC-WG protocols.

11 Notational Conventions

The key words “MUST” “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” are to be interpreted as described in RFC 2119 [1], except that the words do not appear in uppercase.

12 Contributors

Jason Zurawski
Internet2
1150 18th Street, NW
Suite 1020
Washington, DC 20036

D. Martin Swany
University of Delaware
Department of Computer and Information Sciences
Newark, DE 19716

13 Intellectual Property Statement

14 Disclaimer

15 Full Copyright Notice

References

[1] S. Bradner. Key Words for Use in RFCs to Indicate Requirement Levels. RFC 2119, March 1997.

[2] 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.

[3] B. Lowekamp, B. Tierney, L. Cottrell, R. Hughes-Jones, T. Kielmann, and M. Swany. Enabling Network Measurement Portability Through a Hierarchy of Characteristics. In *4th International Workshop on Grid Computing (Grid2003)*, 2003.

[4] A. Anjomshoaa M. Drescher. Standardised Namespaces for XML infosets in OGF. Open grid forum community document, Open Grid Forum, October 2006.

[5] Sun Multi-Schema XML Validator (MSV). `https://msv.dev.java.net/`.

[6] Network Measurement Control Working Group (NMC-WG). `https://forge.gridforum.org/projects/nmc-wg`.

[7] Network Markup Language Working Group (NML-WG). `https://forge.gridforum.org/projects/nml-wg`.

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

[9] RELAX-NG Schema Language. `http://relaxng.org/`.

[10] M. Swany. An Extensible Schema for Network Measurement and Performance Data. Open grid forum working group document, Open Grid Forum, February 2008.

[11] Multi-format schema converter based on RELAX NG. `http://www.thaiopensource.com/relaxng/trang.html`.

[12] XML Schema). `http://www.w3.org/XML/Schema`.

[13] J. Zurawski, M. Swany, and D. Gunter. A scalable framework for representation and exchange of network measurements. In *IEEE/Create-Net Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, Barcelona, Spain, March 2006.