GFD-R-P.087
Category: Recommendation
OGSA ByteIO WG

Editor: Mark Morgan, University of Virginia
31 October 2006
http://forge.gridforum.org/projects/byteio-wg

ByteIO Specification 1.0

Status of This Memo

This memo provides information to the Grid community on efficient manipulation of, access to, and management of bulk data sources and sinks in the grid. Distribution is unlimited.

Abstract

The ByteIO Specification is a description of a set of port types that give users a concise, standard way of interacting with bulk data sources and sinks in the grid. The purpose of these port types is to provide the means for treating such data resources as *POSIX-like* files. At the same time, clients will be able to leverage these port types to provide users with a convenient way of interacting with these grid resources. The purpose of this specification is to address a common case and common requirement in the grid community. Other applications may choose to provide more application specific interfaces for accessing and modifying bulk data in their own resource endpoints, however it is hoped that they will choose to additionally support ByteIO as a means of providing a common interface to which arbitrary clients can speak.

ByteIO is divided into two port types and each addresses a unique set of use cases. The first of these port types supports the notion that a data resource is directly accessible and that clients can handle the maintenance of any session state (such as file pointer, buffering, caching, etc.). The other port type presents a more stream-like interface to clients and as such contains implicit session state. In this latter case data resources with this port type don't represent that bulk data source/sink directly but rather represent the resource of the open stream between the client and the data source/sink.

Contents

1. Introduction

One of the most important challenges facing the grid community at large will be that of achieving buy-in and adoption from potential grid users. Historically, while many grids have solved numerous technical issues, grid users as a whole have felt that the technology failed to address their usability concerns. This perception stems from two inescapable facts:

1) The majority of grid users would prefer that the grid provide them the benefits of thousands of available distributed resources without requiring them to learn new methods and techniques for utilizing those resources;
2) A large number of potential, high-throughput grid applications are legacy applications such as BLAST and SEQUEST which are unaware of the existence of the Grid.

Both of these concerns can be addressed via *access transparency* (one of the core distributed systems transparencies). Users or clients of a distributed system or grid should be unaware (or at least, should not be required to have knowledge of) the distributed nature of their resources. A successful grid should provide the means to allow potential grid clients to use the grid without requiring an intimate knowledge of the nature of the grid.

Access transparency should manifest itself in all facets of the grid ranging from compute resources (clients should not be required to have knowledge of the grid in order to launch processes on it), to security (i.e. single sign-on, etc.), to file systems (access to data should be agnostic of data location in the grid). It is this last item that the byteio-wg addresses. In particular, the byteio-wg's goals are to develop and recommend a set of service port types that would enable *POSIX-like* access to grid data resources and in doing so ease the burden of using such resources in grid applications. In the limit, these services should allow for various file access paradigms and protocols (such as NFS, CIFS, FTP, etc.) to be implemented which will completely (and efficiently) hide the gory details of the grid from users and clients which wish (or are by design) unaware of the nature of the grid. Additionally, a *POSIX-like* interface is familiar to a large audience of potential grid users and grid developers, thus promoting ease of use and implementation and aiding in adoption.

1.1 Outline for this Document

The remainder of this document will be organized as follows. First, we will present a high level overview of the port types we recommend for the ByteIO specification. We will follow this with sections which drill down into the details for each of the port types. Born of the necessity to support potentially numerous OGSA Basic Profiles (of which at the time of this document's writing, only one such profile exists – the OGSA WSRF Basic Profile 1.0 [**WSRFProfileDoc**]), explicit WSDL cannot be given, but a pseudo-schema for the port types will be indicated where applicable[1]. Finally, we will summarize the information in this document and wrap up with information about security considerations, author information, and glossary terms. Accompanying this document will be a number of **Profile Rendering** documents which will normatively describe the details as they pertain to the OGSA Basic Profile in question.

[1] In order to give normative specifications for various port types and in light of this requirement that OGSA specifications are referent to basic profiles of a diverse nature, it seems obvious that any specification will need to be accompanied by various *rendering* documents which will describe normatively how to map the basic port types to the various profiles.

1.2 Terminology

The keywords “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, “OPTIONAL” in this document are to be interpreted as described in [**RFC 2119**].

In addition to the terms introduced in [**RFC 2119**], additional terms commonly used in this document are defined in the Glossary in the back.

When describing abstract data models, this specification uses the notational convention used by the [**XML Infoset**].

When describing concrete XML schemas, this specification uses the notational convention of [**WS-Security**]. Specifically, each member of an element’s [children] or [attributes] property is described using an XPath-like notation (e.g., /x:MyHeader/x:SomeProperty/@value1). The use of {any} indicates the presence of an element wildcard (<xsd:any/>). The use of @{any} indicates the presence of an attribute wildcard (<xsd:anyAttribute/>).

1.3 Namespaces

The following namespaces are used in this document:

Prefix	Namespace
s11	http://schemas.xmlsoap.org/soap/envelope
xsd	http://www.w3.org/2001/XMLSchema
wsa	http://www.w3.org/2005/08/addressing
byteio	http://schemas.ggf.org/byteio/2005/10/byte-io
rbyteio	http://schemas.ggf.org/byteio/2005/10/random-access
sbyteio	http://schemas.ggf.org/byteio/2005/10/streamable-access

2. ByteIO Port Types

ByteIO is divided into two separate and distinct port types – each addressing a unique set of use cases. The first of these port types supports the notion that a data resource is directly accessible and that clients can handle the maintenance of any session state. This port type is specifically designed to ease the burden of service authoring by pushing off much of the management to the client libraries. The client manages things like size, position, etc. The other port type is useful for clients that wish for more session-able semantics in their data interactions. In this latter case resources with this port type don’t represent that bulk data source/sink directly so much as an open session between the client and the data. It is expected that many implementations of either port type will wrap implementations of the other. For example, a stream ByteIO resource could be a session between a client and a non-stream ByteIO resource (henceforth referred to as a Random ByteIO resource).

2.1 Bulk Data Transfer Mechanisms

One of the goals for the ByteIO Working Group was to develop a simple specification to transfer large amounts of bulk data efficiently. While [**SOAP 1.1**] is a reasonable communication medium for describing many types of data, its use as a means of efficient bulk data transfer is questionable at best. However, at the same time, not all consumers or producers can be expected to support more complicated transfer mechanisms. As a compromise, any message which requires the transfer of bulk data takes as a parameter a URI which describes the desired transfer mechanism. At the same time, ByteIO resources will advertise which bulk data transfer

mechanisms they can support. In this way, clients will be able to potentially choose the transfer mechanism which suits their specific needs the best.

At this time the following transfer mechanisms are documented:

- http://schemas.ggf.org/byteio/2005/10/transfer-mechanisms/simple
- http://schemas.ggf.org/byteio/2005/10/transfer-mechanisms/dime
- http://schemas.ggf.org/byteio/2005/10/transfer-mechanisms/mtom

The latter two transfer mechanisms are straightforward utilizations of their respective specifications (i.e., [**DIME**] and [**MTOM**]). The former implies a very simple transfer where the raw data is in-fact included in the applicable [**SOAP 1.1**] messages as a [**Base64**] encoded text element[2]. To ensure that all clients can communicate with all ByteIO resources, a ByteIO implementation MUST support at least the http://schemas.ggf.org/byteio/2005/10/byte-io/.transfer-mechanisms/simple transfer mechanism. It MAY additionally support any other available transfer.

The following two URIs are reserved for future expansion of the specification to include various flavors of HTTP based data transfer:

- http://schemas.ggf.org/byteio/2005/10/transfer-mechanisms/http
- http://schemas.ggf.org/byteio/2005/10/transfer-mechanisms/parallel-http

In order to support this level of flexibility in the bulk data transfer operations, any message which involves the transport of bulk data information (be it a read or a write, a request or a response) MUST include the following XML element as a bulk transfer payload[3]:

```
<byteio:transfer-information-type transfer-mechanism=”xsd:anyURI”>
        {any}*
</byteio:transfer-information-type>
```

The components of the transfer-information-type data type are further described as follows:

/byteio:transfer-information-type/@transfer-mechanism
The URI which names the transfer mechanism in use (as defined above).

Further interpretation of the xsd:any element inside of this element is to be “profiled” later to match various transfer mechanisms (see attached appendices at the end of this document for normative descriptions of the given transfer mechanisms).

2.2 Short Reads

All read/write operations in the ByteIO port types follow the C#/Java semantics for short reads and complete writes. In other words, all read operations are permitted to return less bytes than the number requested. This is to support ByteIO sources which may not want to block on unavailable data or for which the amount of data simply doesn’t satisfy completely the requested amount. This is not an error condition and no faults should be generated. It is however the case that the operations should never return 0 bytes unless the end of the resource has been reached

[2] See **Appendix C** for a normative description of this transfer mechanism.
[3] Note that this element is included in all messages even if normally one wouldn’t consider that message to contain bulk data (for example, this data item is part of the result message from write operations even though the bulk data is conceptually only transferred during the request. This symmetry is to support complicated out-of-band bulk data transfer protocols which require more advanced handshaking. Each transfer mechanism “profile” is expected to fully describe the nature of these messages.

or specified. However, write operations must always completely write all data requested. Failing to do so is an error and an appropriate fault should be generated.

2.3 RandomByteIO Interface

The RandomByteIO port allows clients to access bulk data sources in a session-less, random way, much like the back end of a local file system would work – in other words, clients ask to read or write blocks of data starting at given offsets. The RandomByteIO interface is conceptually defined as follows:

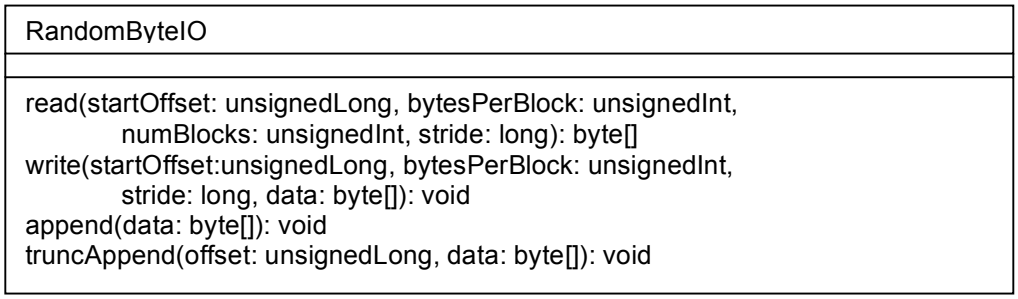

Figure 1: Conceptual Interface for RandomByteIO

2.3.1 RandomByteIO read

The read message is sent to an RandomByteIO implementation when a client wishes to obtain blocks of bulk data within the resource. The RandomByteIO resource MUST respond to this message with a readResponse message but MAY respond with fewer bytes of data then requested. A response with 0 bytes of bulk data indicates that the offset is beyond the limit of the RandomByteIO. This operation MUST happen atomically on the service side of the call.

2.3.1.1 RandomByteIO read

The format of the read Message is:

```
…
<rbyteio:read>
        <rbyteio:start-offset>xsd:unsignedLong</rbyteio:start-offset>
        <rbyteio:bytes-per-block>xsd:unsignedInt</rbyteio:bytes-per-block>
        <rbyteio:num-blocks>xsd:unsignedInt</rbyteio:num-blocks>
        <rbyteio:stride>xsd:long</rbyteio:stride>
        <rbyteio:transfer-information transfer-mechanism=”xsd:anyURI”>
                byteio:transfer-information-type
        </rbyteio:transfer-information>
</rbyteio:read>
…
```

The components of the read message are further described as follows:

/rbyteio:start-offset

The offset (an unsigned long describing the number of bytes) into the data resource at which the client wishes to begin reading[4].

/rbyteio:bytes-per-block
The number of bytes in a single block that the client wishes to read

/rbyteio:num-blocks
The number of blocks that the client is reading

/rbyteio:stride
The offset or delta describing how far apart the beginnings of each block of data are inside the data source.

/rbyteio:transfer-information
A bulk transfer information block as described above in section 2.1 which contains information about the transfer-mechanism being used (and possibly the data itself as per the transfer mechanism).

/rbyteio:transfer-information/@transfer-mechanism
A URI which describes which transfer mechanism the client is using to send this message. The RandomByteIO resource MAY refuse to process this write request if the transfer mechanism used is not supported by the resource.

The response to the read message is a message of the following form:

```
…
<rbyteio:readResponse>
        <rbyteio:transfer-information transfer-mechanism=”xsd:anyURI”>
                byteio:transfer-information-type
        </rbyteio:transfer-information>
</rbyteio:readResponse>
…
```

The components of the readResponse message are further described as follows:

/rbyteio:transfer-information
The transfer information data (as described above in section 2.1) which contains either the actual resultant data or information about how to retrieve the bulk data (as per the transfer mechanism specified).

/rbyteio:transfer-information/@transfer-mechanism
A URI describing the transfer mechanism that is being employed. This transfer mechanism MUST match that requested by the client in the read message.

2.3.1.2 Example SOAP Encoding of the read Message Exchange

The following is a non-normative example of a read request message using [**SOAP 1.1**]:

```
<s11:Envelope
        xmlns:s11=”http://www.w3.org/2003/05/soap-envelope”
```

[4] A method to read from negative offsets (i.e. read from the end of the file) is not provided. It is assumed that the client can calculate the appropriate offset from the total file size. The model for

```
        xmlns:wsa="http://www.w3.org/2005/08/addressing"
        xmlns:byteio="http://schemas.ggf.org/byteio/2005/10/byte-io"
        xmlns:rbyteio="http://schemas.ggf.org/byteio/2005/10/random-access">
  <s11:Header>
        <wsa:Action>
              http://schemas.ggf.org/byteio/2005/10/random-access/read
        </wsa:Action>
        <wsa:To s11:mustUnderstand="1">
              http://www.byteio.org/RandomByteIOSource
        </wsa:To>
  </s11:Header>

  <s11:Body>
        <rbyteio:read>
              <rbyteio:start-offset>1024</rbyteio:start-offset>
              <rbyteio:bytes-per-block>512</rbyteio:bytes-per-block>
              <rbyteio:num-blocks>4</rbyteio:num-blocks>
              <rbyteio:stride>1024</rbyteio:stride>
              <rbyteio:transfer-information
                transfer-mechanism="http://schemas.ggf.org/byteio/2005/10/transfer-
mechanisms/dime">
              </rbyteio:transfer-information>
        </rbyteio:read>
  </s11:Body>
</s11:Envelope>
```

The following is a non-normative example of a read response message using [**SOAP 1.1**]:

```
<s11:Envelope
        xmlns:s11="http://www.w3.org/2003/05/soap-envelope"
        xmlns:wsa="http://www.w3.org/2005/08/addressing"
        xmlns:byteio="http://schemas.ggf.org/byteio/2005/10/byte-io"
        xmlns:rbyteio="http://schemas.ggf.org/byteio/2005/10/random-access">
  <s11:Header>
        <wsa:Action>
              http://schemas.ggf.org/byteio/2005/10/random-access/readResponse
        </wsa:Action>
        <wsa:To>
              http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous
        </wsa:To>
  </s11:Header>

  <s11:Body>
        <rbyteio:readResponse>
              <rbyteio:transfer-information
                transfer-mechanism="http://schemas.ggf.org/byteio/2005/10/transfer-
mechanisms/dime">
              </rbyteio:transfer-information>
        </rbyteio:readResponse>
  </s11:Body>
</s11:Envelope>
```

RandomByteIO in general assumes that the web service is simple and client is complex.

2.3.2 RandomByteIO write

The write message is sent to an RandomByteIO implementation when a client wishes to set bulk data within the resource. The RandomByteIO resource MUST respond to this message with a writeResponse message. This operation MUST happen atomically on the service side of the call.

2.3.2.1 RandomByteIO write

The format of the write message is:

```
…
<rbyteio:write>
        <rbyteio:start-offset>xsd:unsignedLong</rbyteio:start-offset>
        <rbyteio:bytes-per-block>xsd:unsignedInt</rbyteio:bytes-per-block>
        <rbyteio:stride>xsd:long</rbyteio:stride>
        <rbyteio:transfer-information transfer-mechanism=”xsd:anyURI”>
                byteio:transfer-information-type
        </rbyteio:transfer-information>
</rbyteio:write>
…
```

The components of the write message are further described as follows:

/rbyteio:start-offset
: The offset into the RandomByteIO resource at which to begin writing the block of data.

/rbyteio:bytes-per-block
: The number of bytes of data that are to be written for each block of data.

/rbyteio:stride
: The number of bytes that separate the beginnings of each block in the data sink. Blocks are considered to be written sequentially in the order indicated by the stride (i.e., if the stride is positive, then the sequence is in ascending absolute offset order whereas if the stride is negative, the presumed order is descending absolute offset of the blocks). The implication here is that the offsets of the blocks are calculated in isolation from the actual block sizes. If the stride is larger then the block size, then the blocks will be written to the file leaving holes in the middle (the behavior being that these holes should retain any values present prior to the write request if applicable, otherwise the values in the holes are undefined). If the stride is less then the block size, then blocks will overlap and the final value of any given written byte will be that of the last block in the sequence to overlap that byte. Regardless of the stride value, blocks are assumed to be concatenated directly together in the write requests data block (i.e., stride does not apply to this block).

/rbyteio:transfer-information
: A transfer information block as described above in section 2.1 which contains information about the transfer-mechanism being used (and possibly the data itself as per the transfer mechanism).

/rbyteio:transfer-information/@transfer-mechanism
: A URI which describes which transfer mechanism the client is using to send this message. The RandomByteIO resource MAY refuse to process this write request if the transfer mechanism used is not supported by the resource.

The response to the write message is a message of the following form:

```
…
<rbyteio:writeResponse>
        <rbyteio:transfer-information transfer-mechanism=”xsd:anyURI”>
                byteio:transfer-information-type
        </rbyteio:transfer-information>
</rbyteio:writeResponse>
…
```

The components of the writeResponse message are further described as follows:

/rbyteio:transfer-information
A transfer information block as described above in section 2.1 which contains information about the transfer-mechanism being used (and possibly the data itself as per the transfer mechanism).

/rbyteio:transfer-information/@transfer-mechanism
A URI which describes which transfer mechanism the client is using to send this message. The RandomByteIO resource MAY refuse to process this write request if the transfer mechanism used is not supported by the resource.

2.3.2.2 Example SOAP Encoding of the write Message Exchange

The following is a non-normative example of a write message using [**SOAP 1.1**]

```
<s11:Envelope
        xmlns:s11=”http://www.w3.org/2003/05/soap-envelope”
        xmlns:wsa=”http://www.w3.org/2005/08/addressing”
        xmlns:byteio=”http://schemas.ggf.org/byteio/2005/10/byte-io”
        xmlns:rbyteio=”http://schemas.ggf.org/byteio/2005/10/random-access”>
  <s11:Header>
        <wsa:Action>
                http://schemas.ggf.org/byteio/2005/10/random-access/write
        </wsa:Action>
        <wsa:To s11:mustUnderstand=”1”>
                http://www.byteio.org/RandomByteIOSource
        </wsa:To>
  </s11:Header>

  <s11:Body>
        <rbyteio:write>
                <rbyteio:start-offset>1024</rbyteio:start-offset>
                <rbyteio:bytes-per-block>512</rbyteio:bytes-per-block>
                <rbyteio:stride>1024</rbyteio:stride>
                <rbyteio:transfer-information transfer-
mechanism="http://schemas.ggf.org/byteio/2005/10/byte-io/transfer-mechanisms/dime”>
                </rbyteio:transfer-information>
        </rbyteio:write>
  </s11:Body>
</s11:Envelope>
```

The following is a non-normative example of a write response message using [**SOAP 1.1**]:

```
<s11:Envelope
        xmlns:s11="http://www.w3.org/2003/05/soap-envelope"
        xmlns:wsa="http://www.w3.org/2005/08/addressing"
        xmlns:byteio="http://schemas.ggf.org/byteio/2005/10/byte-io"
        xmlns:rbyteio="http://schemas.ggf.org/byteio/2005/10/random-access">
  <s11:Header>
        <wsa:Action>
                http://schemas.ggf.org/byteio/2005/10/random-access/writeResponse
        </wsa:Action>
        <wsa:To>
                http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous
        </wsa:To>
  </s11:Header>

  <s11:Body>
        <rbyteio:writeResponse>
                <rbyteio:transfer-information transfer-
mechanism="http://schemas.ggf.org/byteio/2005/10/transfer-mechanisms/dime">
                </rbyteio:transfer-information>
        </rbyteio:writeResponse>
  </s11:Body>
</s11:Envelope>
```

2.3.3 RandomByteIO append

The append message is sent to an RandomByteIO implementation when a client wishes to append a block of bulk data to the end of an RandomByteIO resource. The RandomByteIO resource MUST respond to this message with an appendResponse message. This operation MUST happen atomically on the service side of the call.

2.3.3.1 RandomByteIO append

The format of the append message is:

```
…
<rbyteio:append>
        <rbyteio:transfer-information transfer-mechanism="xsd:anyURI">
                byteio:transfer-information-type
        </rbyteio:transfer-information>
</rbyteio:append>
…
```

The components of the append message are further described as follows:

/rbyteio:transfer-information
: A transfer information block as described above in section 2.1 which contains information about the transfer-mechanism being used (and possibly the data itself as per the transfer mechanism).

/rbyteio:transfer-information/@transfer-mechanism
: A URI which describes which transfer mechanism the client is using to send this message. The RandomByteIO resource MAY refuse to process this append request if the transfer mechanism used is not supported by the resource.

The response to the append message is a message of the following form:

```
…
<rbyteio:appendResponse>
        <rbyteio:transfer-information transfer-mechanism=”xsd:anyURI”>
                byteio:transfer-information-type
        </rbyteio:transfer-information>
</rbyteio:appendResponse>
…
```

The components of the appendResponse message are further described as follows:

/rbyteio:transfer-information
A transfer information block as described above in section 2.1 which contains information about the transfer-mechanism being used (and possibly the data itself as per the transfer mechanism).

/rbyteio:transfer-information/@transfer-mechanism
A URI which describes which transfer mechanism the client is using to send this message. The RandomByteIO resource MAY refuse to process this append request if the transfer mechanism used is not supported by the resource.

2.3.3.2 Example SOAP Encoding of the append Message Exchange

The following is a non-normative example of an append message using [**SOAP 1.1**]

```
<s11:Envelope
        xmlns:s11=”http://www.w3.org/2003/05/soap-envelope”
        xmlns:wsa=”http://www.w3.org/2005/08/addressing”
        xmlns:byteio=”http://schemas.ggf.org/byteio/2005/10/byte-io”
        xmlns:rbyteio=”http://schemas.ggf.org/byteio/2005/10/random-access”>
  <s11:Header>
        <wsa:Action>
                http://schemas.ggf.org/byteio/2005/10/random-access/append
        </wsa:Action>
        <wsa:To s11:mustUnderstand=”1”>
                http://www.byteio.org/RandomByteIOSource
        </wsa:To>
  </s11:Header>

  <s11:Body>
        <rbyteio:append>
                <rbyteio:transfer-information transfer-
mechanism="http://schemas.ggf.org/byteio/2005/10/byte-io/transfer-mechanisms/dime”>
                </rbyteio:transfer-information>
        </rbyteio:append>
  </s11:Body>
</s11:Envelope>
```

The following is a non-normative example of an append response message using [**SOAP 1.1**]:

```
<s11:Envelope
        xmlns:s11=”http://www.w3.org/2003/05/soap-envelope”
        xmlns:wsa=”http://www.w3.org/2005/08/addressing”
```

```
        xmlns:byteio="http://schemas.ggf.org/byteio/2005/10/byte-io"
        xmlns:rbyteio="http://schemas.ggf.org/byteio/2005/10/random-access">
  <s11:Header>
        <wsa:Action>
              http://schemas.ggf.org/byteio/2005/10/random-access/appendResponse
        </wsa:Action>
        <wsa:To>
              http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous
        </wsa:To>
  </s11:Header>

  <s11:Body>
        <rbyteio:appendResponse>
        <rbyteio:transfer-information transfer-
mechanism="http://schemas.ggf.org/byteio/2005/10/byte-io/transfer-mechanisms/dime">
              </rbyteio:transfer-information>
        </rbyteio:appendResponse>
  </s11:Body>
</s11:Envelope>
```

2.3.4 RandomByteIO truncAppend

The truncAppend message is sent to an RandomByteIO implementation when a client wishes to truncate an RandomByteIO resource's bulk data to a given offset and optionally also append a block of bulk data to the end of the truncated RandomByteIO resource. The RandomByteIO resource MUST respond to this message with a truncAppendResponse message. The client MAY choose to truncate to any valid offset within the resource (including 0). The client MAY also choose to append either 0 bytes of data, or a non-zero sized block of data, to the end of the truncated resource. This operation MUST happen atomically on the service side of the call.

2.3.4.1 RandomByteIO truncAppend

The format of the truncAppend message is:

```
…
<rbyteio:truncAppend>
        <rbyteio:offset>xsd:unsignedLong</rbyteio:offset>
        <rbyteio:transfer-information transfer-mechanism="xsd:anyURI">
              byteio:transfer-information-type
        </rbyteio:transfer-information>
</rbyteio:truncAppend>
…
```

The components of the truncAppend message are further described as follows:

/rbyteio:offset
: An offset into the resource to which the resource is to be truncated.

/rbyteio:transfer-information
: A transfer information block as described above in section 2.1 which contains information about the transfer-mechanism being used (and possibly the data itself as per the transfer mechanism).

/rbyteio:transfer-information/@transfer-mechanism

A URI which describes which transfer mechanism the client is using to send this message. The RandomByteIO resource MAY refuse to process this append request if the transfer mechanism used is not supported by the resource.

The response to the truncAppend message is a message of the following form:

```
…
<rbyteio:truncAppendResponse>
        <rbyteio:transfer-information transfer-mechanism=”xsd:anyURI”>
                byteio:transfer-information-type
        </rbyteio:transfer-information>
</rbyteio:truncAppendResponse>
…
```

The components of the truncAppend message are further described as follows:

/rbyteio:transfer-information
A transfer information block as described above in section 2.1 which contains information about the transfer-mechanism being used (and possibly the data itself as per the transfer mechanism).

/rbyteio:transfer-information/@transfer-mechanism
A URI which describes which transfer mechanism the client is using to send this message. The RandomByteIO resource MAY refuse to process this append request if the transfer mechanism used is not supported by the resource.

2.3.4.2 Example SOAP Encoding of the truncAppend Message Exchange

The following is a non-normative example of a truncAppend message using [**SOAP 1.1**]

```
<s11:Envelope
        xmlns:s11=”http://www.w3.org/2003/05/soap-envelope”
        xmlns:wsa=”http://www.w3.org/2005/08/addressing”
        xmlns:byteio=”http://schemas.ggf.org/byteio/2005/10/byte-io”
        xmlns:rbyteio=”http://schemas.ggf.org/byteio/2005/10/random-access”>
  <s11:Header>
        <wsa:Action>
                http://schemas.ggf.org/byteio/2005/10/random-access/truncAppend
        </wsa:Action>
        <wsa:To s11:mustUnderstand=”1”>
                http://www.byteio.org/RandomByteIOSource
        </wsa:To>
  </s11:Header>

  <s11:Body>
        <rbyteio:truncAppend>
                <rbyteio:offset>0</rbyteio:offset>
                <rbyteio:transfer-information transfer-
mechanism="http://schemas.ggf.org/byteio/2005/10/byte-io/transfer-mechanisms/dime”>
                </rbyteio:transfer-information>
        </rbyteio:truncAppend>
  </s11:Body>
</s11:Envelope>
```

The following is a non-normative example of a truncAppend response message using [**SOAP 1.1**]:

```
<s11:Envelope
        xmlns:s11="http://www.w3.org/2003/05/soap-envelope"
        xmlns:wsa="http://www.w3.org/2005/03/addressing"
        xmlns:byteio="http://schemas.ggf.org/byteio/2005/10/byte-io"
        xmlns:rbyteio="http://schemas.ggf.org/byteio/2005/10/random-access">
  <s11:Header>
        <wsa:Action>
                http://schemas.ggf.org/byteio/2005/10/random-
access/truncAppendResponse
        </wsa:Action>
        <wsa:To>
                http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous
        </wsa:To>
  </s11:Header>

  <s11:Body>
        <rbyteio:truncAppendResponse>
        <rbyteio:transfer-information transfer-
mechanism="http://schemas.ggf.org/byteio/2005/10/byte-io/transfer-mechanisms/dime">
                </rbyteio:transfer-information>
        </rbyteio:truncAppendResponse>
  </s11:Body>
</s11:Envelope>
```

2.4 StreamableByteIO Interface

The StreamableByteIO port allows clients to access bulk data sources via a stateful session resource – in other words, clients will open (through means not normatively described by ByteIO) a session resource to a data source/sink and will then read and write to and from that stream as required. The StreamableByteIO interface is conceptually defined as follows:

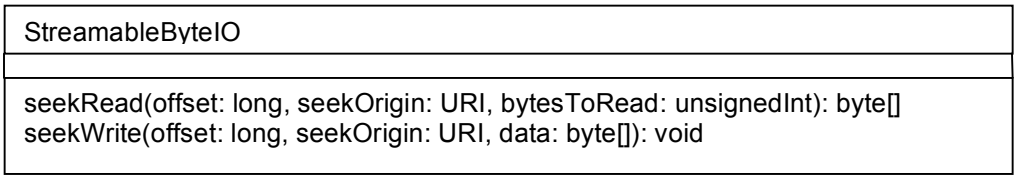

Figure 2: Conceptual Interface for StreamableByteIO

2.4.1 StreamableByteIO seekRead

The seekRead message is sent to an StreamableByteIO implementation when a client wishes to read a block of data from the resource. This is combined with the notion of a seek operation to allow for a smaller number of messages to be sent in the common case for a seek (the common case being that almost all seek requests are immediately followed by read or write requests). Note that both singleton seeks (without reads or writes) and non-seek reads and writes are available with this interface by filling in appropriate values for the seek parameters and/or read/write parameters. For a non-seekable stream, the offset for the seek operations MUST be 0, and the seekOrigin MUST indicate the current position (as given below). Failure to follow these guidelines for non-seekable streams SHOULD result in faults being thrown. The StreamableByteIO resource MUST respond to this message with a seekReadResponse

message. The StreamableByteIO resource MAY choose to fail on this message exchange if it does not support the read operation. The resource MAY also response with fewer bytes of data then the client asked for (0 returned bytes indicating that the end of the stream has been reached). This operation MUST happen atomically on the service side of the call.

2.4.1.1 StreamableByteIO seekRead

The format of the seekRead message is:

```
…
<sbyteio:seekRead>
        <sbyteio:offset>xsd:long</sbyteio:offset>
        <sbyteio:seek-origin> xsd:anyURI </sbyteio:seek-origin>
        <sbyteio:num-bytes>xsd:unsignedInt</sbyteio:num-bytes>
        <sbyteio:transfer-information transfer-mechanism=”xsd:anyURI”>
                byteio:transfer-information-type
        </sbyteio:transfer-information>
</sbyteio:seekRead>
…
```

The components of the seekRead message are further described as follows:

/sbyteio:offset
: The offset into the StreamableByteIO resource which the client wishes to seek to.

/sbyteio:seek-origin
: A URI which indicates the origin of this seek operation. Valid URIs include:
  - http://schemas.ggf.org/byteio/2005/10/streamable-access/seek-origins/current -- instructs the StreamableByteIO resource to seek from the current position within the bulk data source/sink.
  - http://schemas.ggf.org/byteio/2005/10/streamable-access/seek-origins/beginning -- instructs the StreamableByteIO resource to seek from the beginning of the bulk data source/sink.
  - http://schemas.ggf.org/byteio/2005/10/streamable-access/seek-origins/end -- instructs the StreamableByteIO resource to seek from the end of the bulk data source/sink.

/sbyteio:num-bytes
: The maximum number of bytes that the client wishes to read from the stream resource.

/sbyteio:transfer-information
: The transfer information data (as described above in section 2.1) which contains either the actual resultant data or information about how to retrieve the bulk data.

/sbyteio:transfer-information/@transfer-mechanism
: A URI describing the transfer mechanism that is being employed.

The response to the seekRead message is a message of the following form:

```
…
<sbyteio:seekReadResponse>
        <sbyteio:transfer-information transfer-mechanism=”xsd:anyURI”>
                byteio:transfer-information-type
        </sbyteio:transfer-information>
```

```
</sbyteio:seekReadResponse>
…
```

The components of the seekReadResponse message are further described as follows:

/sbyteio:transfer-information
: The transfer information data (as described above in section 2.1) which contains either the actual resultant data or information about how to retrieve the bulk data.

/sbyteio:transfer-information/@transfer-mechanism
: A URI describing the transfer mechanism that is being employed. This transfer mechanism MUST match that requested by the client in the seekRead message.

2.4.1.2 Example SOAP Encoding of the seekRead Message Exchange

The following is a non-normative example of a seekRead message using [**SOAP 1.1**]

```
<s11:Envelope
        xmlns:s11="http://www.w3.org/2003/05/soap-envelope"
        xmlns:wsa="http://www.w3.org/2005/08/addressing"
        xmlns:byteio="http://schemas.ggf.org/byteio/2005/10/byte-io"
        xmlns:sbyteio="http://schemas.ggf.org/byteio/2005/10/streamable-access">
  <s11:Header>
        <wsa:Action>
                http://schemas.ggf.org/byteio/2005/10/streamable-access/seek-read
        </wsa:Action>
        <wsa:To s11:mustUnderstand="1">
                http://www.byteio.org/StreamableByteIOSource
        </wsa:To>
  </s11:Header>

  <s11:Body>
        <sbyteio:seekRead>
                <sbyteio:offset>0</sbyteio:offset>
                <sbyteio:seek-origin>http://schemas.ggf.org/byteio/2005/10/streamable-
access/seek-origins/current</sbyteio:seek-origin>
                <sbyteio:num-bytest>1024</sbyteio:num-bytes>
                <sbyteio:transfer-information
                        transfer-
mechanism="http://schemas.ggf.org/byteio/2005/10/transfer-mechanisms/dime">
                 </sbyteio:transfer-information>
        </sbyteio:seekRead>
  </s11:Body>
</s11:Envelope>
```

The following is a non-normative example of a read response message using [**SOAP 1.1**]:

```
<s11:Envelope
        xmlns:s11="http://www.w3.org/2003/05/soap-envelope"
        xmlns:wsa="http://www.w3.org/2005/08/addressing"
        xmlns:byteio="http://schemas.ggf.org/byteio/2005/10/byte-io"
        xmlns:sbyteio="http://schemas.ggf.org/byteio/2005/10/streamable-access">
  <s11:Header>
        <wsa:Action>
```

```
                http://schemas.ggf.org/byteio/2005/10/streamable-access/readResponse
        </wsa:Action>
        <wsa:To>
                http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous
        </wsa:To>
  </s11:Header>

  <s11:Body>
        <sbyteio:readResponse>
            <sbyteio:transfer-information
                transfer-mechanism="http://schemas.ggf.org/byteio/2005/10/transfer-
mechanisms/dime">
            </sbyteio:transfer-information>
        </sbyteio:readResponse>
  </s11:Body>
</s11:Envelope>
```

2.4.2 StreamableByteIO seekWrite

The seekWrite message is sent to an StreamableByteIO implementation when a client wishes to write a block of data tothe resource. This is combined with the notion of a seek operation to allow for a smaller number of messages to be sent in the common case for a seek (the common case being that almost all seek requests are immediately followed by read or write requests). Note that both singleton seeks (without reads or writes) and non-seek reads and writes are available with this interface by filling in appropriate values for the seek parameters and/or read/write parameters. For a non-seekable stream, the offset for the seek operations MUST be 0, and the seekOrigin MUST indicate the current position (as given below). Failure to follow these guidelines for non-seekable streams SHOULD result in faults being thrown. The StreamableByteIO resource MUST respond to this message with a seekWriteResponse message. The StreamableByteIO resource MAY choose to fail on this message exchange if it does not support the write operation. This operation MUST happen atomically on the service side of the call.

2.4.2.1 StreamableByteIO seekWrite

The format of the seekWrite message is:

```
…
<sbyteio:seekWrite>
        <sbyteio:offset>xsd:long</sbyteio:offset>
        <sbyteio:seek-origin> xsd:anyURI </sbyteio:seek-origin>
        <sbyteio:transfer-information transfer-mechanism="xsd:anyURI">
                byteio:transfer-information-type
        </sbyteio:transfer-information>
</sbyteio:seekWrite>
…
```

The components of the seekWrite message are further described as follows:

/sbyteio:offset
: The offset into the StreamableByteIO resource which the client wishes to seek to.

/sbyteio:seek-origin

A URI which indicates the origin of this seek operation. Valid URIs include:

- http://schemas.ggf.org/byteio/2005/10/streamable-access/seek-origins/current -- instructs the StreamableByteIO resource to seek from the current position within the bulk data source/sink.
- http://schemas.ggf.org/byteio/2005/10/streamable-access/seek-origins/beginning -- instructs the StreamableByteIO resource to seek from the beginning of the bulk data source/sink.
- http://schemas.ggf.org/byteio/2005/10/streamable-access/seek-origins/end -- instructs the StreamableByteIO resource to seek from the end of the bulk data source/sink.

/sbyteio:transfer-information
The transfer information data (as described above in section 2.1) which contains either the actual resultant data or information about how to retrieve the bulk data.

/sbyteio:transfer-information/@transfer-mechanism
A URI describing the transfer mechanism that is being employed.

The response to the read message is a message of the following form:

```
…
<sbyteio:seekWriteResponse>
        <sbyteio:transfer-information transfer-mechanism=”xsd:anyURI”>
                byteio:transfer-information-type
        </sbyteio:transfer-information>
</sbyteio:seekWriteResponse>
…
```

The components of the seekWrite message are further described as follows:

/sbyteio:transfer-information
The transfer information data (as described above in section 2.1) which contains either the actual resultant data or information about how to retrieve the bulk data.

/sbyteio:transfer-information/@transfer-mechanism
A URI describing the transfer mechanism that is being employed.

2.4.2.2 Example SOAP Encoding of the seekWrite Message Exchange

The following is a non-normative example of a seekWrite message using [**SOAP 1.1**]

```
<s11:Envelope
        xmlns:s11=”http://www.w3.org/2003/05/soap-envelope”
        xmlns:wsa=”http://www.w3.org/2005/08/addressing”
        xmlns:byteio=”http://schemas.ggf.org/byteio/2005/10/byte-io”
        xmlns:sbyteio=”http://schemas.ggf.org/byteio/2005/10/streamable-access”>
  <s11:Header>
        <wsa:Action>
                http://schemas.ggf.org/byteio/2005/10/streamable-access/seekWrite
        </wsa:Action>
        <wsa:To s11:mustUnderstand=”1”>
                http://www.byteio.org/StreamableByteIOSource
        </wsa:To>
  </s11:Header>
```

```
   <s11:Body>
         <sbyteio:seekWrite>
               <sbyteio:offset>0</sbyteio:offset>
               <sbyteio:seek-origin>http://schemas.ggf.org/byteio/2005/10/streamable-
access/seek-origins/current</sbyteio:seek-origin>
               <sbyteio:transfer-information
                     transfer-mechanism="http://schemas.ggf.org/byteio/2005/10/transfer-
mechanisms/dime">
               </sbyteio:transfer-information>
         </sbyteio:seekWrite>
   </s11:Body>
</s11:Envelope>
```

The following is a non-normative example of a seekWrite response message using [**SOAP 1.1**]:

```
<s11:Envelope
         xmlns:s11="http://www.w3.org/2003/05/soap-envelope"
         xmlns:wsa="http://www.w3.org/2005/03/addressing"
         xmlns:byteio="http://schemas.ggf.org/byteio/2005/10/byte-io"
         xmlns:sbyteio="http://schemas.ggf.org/byteio/2005/10/streamable-access">
   <s11:Header>
         <wsa:Action>
               http://schemas.ggf.org/byteio/2005/10/streamable-
access/seekWriteResponse
         </wsa:Action>
         <wsa:To>
               http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous
         </wsa:To>
   </s11:Header>

   <s11:Body>
         <sbyteio:seekWriteResponse>
               <sbyteio:transfer-information
                     transfer-
mechanism="http://schemas.ggf.org/byteio/2005/10/transfer-mechanisms/dime">
               </sbyteio:transfer-information>
         </sbyteio:seekWriteResponse>
   </s11:Body>
</s11:Envelope>
```

3. Concurrency in ByteIO

As a simple, file-like bulk data specification, ByteIO explicitly does not address the more complicated question of concurrency for multiple readers and writers. Any attempt at managing such concurrency is left up to external (and undefined) services and software. Advisory locking could easily be implemented as an additional service port type and would not impact the simplicity of ByteIO. The only concurrency issues that are addressed by ByteIO are atomicity issues revolving around multiple read/write requests within the service. At that level, each ByteIO specific operation (read, write, append, etc.) should be atomic.

4. ByteIO Properties

The ability to retrieve various properties for ByteIO resources is an important part of this specification. However, it is not possible to normatively describe those properties in this document. The ByteIO specification is referent to the various OGSA Basic Profiles. Because those various profiles may choose to represent properties in different ways, the normative description of the ByteIO properties must be placed in separate documents which normatively describe how to render this specification according to those profiles. However, in order to maintain as much consistency as possible between renderings, it is desirable to describe non-normatively here what properties a ByteIO resource should contain.

4.1 RandomByteIO Properties

The following table indicates the properties that RandomByteIO resources might have for each **Profile Rendering**. The various ByteIO rendering documents MUST describe normatively how to represent these properties. The *Requirement Level* entry in the table describes whether an RandomByteIO resource MUST have the property, SHOULD have the property, or MAY have the property (as per [**RFC2119**]). This list is by no means exhaustive and implementers are free to add their own properties as they see fit. Neither should the names for properties in this table be taken as absolute – they are merely textual identifiers attached to conceptual properties. The true, normative names for various properties MUST be specified by the appropriate rendering document.

Property	Requirement Level	Description
Size	MUST	The total size (in bytes) of the RandomByteIO resource.
Readable	MUST	A Boolean indicating whether or not the resource will allow clients to read information from this resource (using the read message exchange).
Writeable	MUST	A boolean indicating whether or not the resource will allow the write messages (write, append, and truncAppend).
TransferMechanism	MUST	A list of URI's indicating the transfer mechanisms supported by the ByteIO resource in question. If a transfer mechanism is listed here, then the ByteIO resource MUST support that transfer mechanism as per the appropriate specification.
CreateTime*	MAY	The timestamp for when this resource was created (relative to the hosting environment). Creation time is loosely defined as the time at which a client could first use any of the methods (or access any of the properties) on an RandomByteIO resource and receive a valid, non-error response message. It is left up to the implementer to determine what this time should be.
ModificationTime*	MAY	The timestamp for when this resource was last modified (relative to the hosting environment). Modification time is defined as any time after which any client having previously read a block of data (with the read method), or having accessed the Size or ReadOnly properties, could now expect to receive different results were it to call

* All timestamps are with respect to the hosting environment for the resource. No attempt at a global clock is intended. In other words, the CreateTime, AccessTime, and ModificationTime for a resource all represent timestamps taken by the hosting environment and synchronized only to that hosting environment's clock. No guarantees are made beyond that.

		the read method or access the properties again.
AccessTime*	MAY	The timestamp for when this resource was last accessed (relative to the hosting environment). The last access time is defined as the last time when any of the methods described in the RandomByteIO specification were last called.

4.2 StreamableByteIO Properties

The following table indicates the properties that StreamableByteIO resources might have for each **Profile Rendering**. The various ByteIO rendering documents MUST describe normatively how to represent these properties. The *Requirement Level* entry in the table describes whether an StreamableByteIO resource MUST have the property, SHOULD have the property, or MAY have the property (as per [**RFC2119**]). This list is by no means exhaustive and implementers are free to add their own properties as they see fit.

Property	**Requirement Level**	**Description**
Size	MAY	If available, this describes the current total length of the stream.
Position	SHOULD	This property describes the current position of the stream pointer.
Readable	MUST	A boolean value indicating whether or not reads are allowed to this stream. This value is most likely dependent on both the underlying resource as well as possibly the flags used during creation (opening) of the stream.
Writeable	MUST	A boolean value indicating whether or not writes are allowed to this stream. This value is most likely dependent on both the underlying resource as well as possibly the flags used during creation (opening) of the stream.
Seekable	MUST	A boolean value indicating whether or not non-zero seek values are permitted for this stream.
TransferMechanism	MUST	A list of URI's indicating the transfer mechanisms supported by the ByteIO resource in question. If a transfer mechanism is listed here, then the ByteIO resource MUST support that transfer mechanism as per the appropriate specification.
EndOfStream	MUST	A boolean property which MUST be set to true when the end of the stream has been reached. Once this property becomes true, all attempts to read without first seeking back into the stream must result in 0 bytes being returned to the caller. Note that for writeable streams, this property does not indicate that further writes would fail.
DataResource	MAY	This property, if available, is a WS-Addressing EndpointReferenceType which indicates the data source/sink from/to which the stream is connected. Note that this is largely implementation specific and not all streams will require or even permit a WS-Addressing addressable resource to interact with. On the other hand, some streams will and this information could be very useful to clients when the resource is available.

5. ByteIO Lifetime Management

Lifetime management is another important part of the ByteIO specifications. Unfortunately, for reasons similar to those indicated in the properties section above (Section 3), it is not possible to describe this functionality normatively in this document. Rather, normative descriptions of the port types must be postponed until the appropriate **Profile Rendering** document. However, we describe here the semantics of the lifetime operations as completely as possible.

5.1 Creation

Creation is out of scope for the ByteIO specifications. However, given the close relationship between StreamableByteIO resources and their creation (or opening), the authors of this document felt that it was worthwhile to say a few words about creation at a high level and to suggest some possible interfaces and data types.

For creation of StreamableByteIO resources, we recommend that users be able to identify both an open mode for the stream and an access mode for that stream. The open mode should indicate whether a user wishes to create, append, truncate, create-new, open, or open-or-create the data source/sink. This becomes particularly important in terms of the create-new operation. This operation is used by large numbers of legacy applications to achieve atomicity with respect to implementations for file locking and data grids have been rejected in the past for not correctly supporting this functionality. In terms of file access modes the authors recommend that clients have the ability to specify whether the stream is being opened for read, write, or read-and-write access.

5.2 Destruction

Destruction is very much in scope for ByteIO. As was mentioned above, the normative description of destruction is reserved for **Profile Renderings**, however semantic descriptions of this operation is the same regardless of rendering.

The main difficulty of describing the end of life management functionality for ByteIO comes from the schism between RandomByteIO and StreamableByteIO resources. In the former case, the data source/sink itself is being referenced and manipulated whereas in the latter case, the resource really represents an instantiated access abstraction. For this reason, the semantics of destroy are different for these two resource types.

When an RandomByteIO resource receives the destroy request, subject to security, that resource should be terminated and appropriate clean-up operations performed. What this means is somewhat implementation dependent, but whether the actual data source/sink (a file on disk, an entry in a database, etc.) is destroyed, or merely the web service resource representation of that resource is up to implementers to decide. The only requirement is that destroy operations on RandomByteIO resources should have the affect of ensuring that any further requests sent to that resource's WS-Addressing EndpointReferenceType should fail with an error (rendered using the appropriate Profile Rendering) indicating that the resource no longer exists. For the purposes of this specification, destruction capabilities on RandomByteIO implementations are not required but MAY be included in various implementations with the semantics described above.

Similar to RandomByteIO, StreamableByteIO resources which receive a destroy request should also cause future requests to the same WS-Addressing EndpointReferenceType to fail. However, in this case it is clear that the destruction operation should never destroy the underlying data source/sink (whether that data source/sink be a web service resource or not). This operation should be considered semantically the "close" operation on the stream and nothing

more. An implementation of the StreamableByteIO port type MUST contain a *destroy* or *close* mechanism appropriate to that implementation's **Profile Rendering**.

6. Faults and Failures

As before with properties and lifetime management, it is not possible to normatively describe the faulting and failure mechanisms for ByteIO in this document. Instead in this section we will non-normatively describe the fault and failure conditions in terms of causes and information available to calling clients and leave it up to the **Profile Rendering** documents to normatively describe the exact syntax for conveying the appropriate information.

6.1 Available Faults and Failures

This section describes every possible fault and failure that is relevant to the various ByteIO port types. Following this section we will indicate every message exchange possible between clients and ByteIO resources and list for each the faults and failures that that message exchange might generate.

Resource Unavailable Failure	This failure indicates that a message tried to access a resource which does not exist (at least, not at the address given). No specific information is required of the Profile Rendering. This failure may be generated by any ByteIO message exchange and as such is not individually listed for each operation.
Unsupported Transfer	This failure may be generated by any message exchange with a ByteIO resource that indicates a transfer mechanism not supported by the target. Each Profile Rendering is required to include a list of the supported transfer mechanisms with the information for this failure. This failure may be generated by any ByteIO message exchange and as such is not individually listed for each operation.
Write Not Permitted Failure	This failure is generated any time a client request to write data to a ByteIO resource is denied due to restrictions on that resource (i.e., because the resource is read-only). No specific information is required of the Profile Rendering.
Read Not Permitted Failure	This failure is generated any time a client request to read data from a ByteIO resource is denied due to restrictions on that resource (i.e., because the resource is write-only). No specific information is required of the Profile Rendering.
Truncate Not Permitted Failure	This failure indicates that for whatever reason, the truncate operation can't be permitted. If a truncate fails because writing isn't permitted, then the Write Not Permitted Failure should be raised. However, if writes are allowed and the truncate operation separately is not permitted, then this failure might be raised. No specific information is required of the Profile Rendering.
Seek Not Permitted Failure	This failure indicates that a seek operation was attempted on a stream that doesn't support seeking. To avoid this failure, clients that wish to use seekRead and seekWrite on a resource that doesn't support seeking should send in seek values indicating an offset of 0 bytes, and with the seekOrigin set to the current position value. No specific information is required of the Profile Rendering.
Custom Failure	It isn't possible to identify every failure that might occur in a ByteIO implementation. For example, for some

implementations, you may fail because the disk is full for a backend file system; while in another you could fail because the sensor from which you were retrieving your data suddenly disappears. Because of this, rather then specify an unknown failure type, we wish to indicate with this failure type that implementers are encouraged to add their own failures for special cases as they see fit. This failure may be generated by any ByteIO message exchange and as such is not individually listed for each operation.

6.2 Message Exchange Failures for RandomByteIO

The following describes what failures can be generated by each RandomByteIO message exchange (in addition to those indicated as possible failures for ALL message exchanges).

6.2.1 read

- Read Not Permitted Failure

6.2.2 write

- Write Not Permitted Failure

6.2.3 append

- Write Not Permitted Failure

6.2.4 truncAppend

- Write Not Permitted Failure
- Truncate Not Permitted Failure

6.3 Message Exchange Failures for StreamableByteIO

The following describes what failures can be generated by each StreamableByteIO message exchange (in addition to those indicated as possible failures for ALL message exchanges).

6.3.1 seekRead

- Seek Not Permitted Failure
- Read Not Permitted Failure

6.3.2 seekWrite

- Seek Not Permitted Failure
- Write Not Permitted Failure

7. Security Considerations

Security is of the utmost importance for ByteIO implementations. For many implementations, access to sensitive information is being made available which must be protected. Data integrity of both data sources and sinks as well as the data transferred on the wire must be maintained (when desired by the clients and services) and authentication for the purposes of allowing or denying access to various pieces of information is also relevant.

The ByteIO specification is intended to function using whatever security solutions become available to the web services and OGSA communities. Further, transfer mechanisms should be available which allow for both signing and encryption of data streams as requested.

Because Security is being separately addressed by other working groups within OGSA, the exact details of security in terms of ByteIO are out of scope. However, it should be noted that prototype

implementations of the various ByteIO port types must by necessity be carefully written so as to minimize exposure of sensitive data. Until a security solution becomes available to OGSA, these prototype implementations will have large security holes in the testing environments.

Author Information

Editor:
Mark Morgan
University of Virginia, Department of Computer Science
151 Engineer's Way
P.O. Box 400740
Charlottesville, VA. 22904-4740
Phone: +1 (434) 982-2047
E-mail: mmm2a@cs.virginia.edu

Thanks to Neil P. Chue Hong, Andrew Grimshaw, Allen Luniewski, Michel Drescher, Glenn Wasson and to the SAGA team as a whole for their invaluable input.

Glossary

Conceptual Interface	An interface which describes the conceptual behavior of a service but which doesn't necessarily reflect the actual parameters and methods that are being received and sent.
Profile Rendering	As an OGSA Basic Profile referent specification, ByteIO must by necessity be flexible enough to allow multiple normative definitions (at least one each for each OGSA Basic Profile) while at the same time be described in as much detail and rigor as possible to allow for the greatest chance of interoperability. To satisfy these desires and constraints, the ByteIO specification includes both a general description document (this document) as well as additional "rendering" documents that describe how ByteIO should be realized in the various OGSA Basic Profiles. A profile rendering is one of these normative documents.
OGSA WSRF Basic Profile Rendering	Because various aspects of a specification for a concrete port type may require functionality covered in various OGSA Basic Profiles, each specification will have to be non-normatively described in an initial document and then normatively refined or rendered in a specific profile. This term refers to the rendering of the ByteIO specification in the OGSA WSRF Basic Profile.

Intellectual Property Statement

of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the OGF Secretariat.

The OGF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this recommendation. Please address the information to the OGF Executive Director.

Full Copyright Notice

References

[WSRFProfileDoc] I. Foster, T. Maguire, D. Snelling, *OGSA WSRF Basic Profile 1.0*, https://forge.gridforum.org/projects/ogsa-wg/document/draft-ggf-ogsa-wsrf-basic-profile/en/15, GWS-R (draft-ggf-ogsa-wsrf-basic-profile-021), 6 July 2005.

[RFC2119] S.Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, http://www.ietf.org/rfc/rfc2119.txt, IETF RFC 2119, March 1997.

[XML-Infoset] http://www.w3.org/TR/xml-infoset/

[XPATH] http://www.w3.org/TR/xpath

[WS-Addressing] M. Gudgin, M. Hadley, and T. Rogers (ed.) Web Services Addressing 1.0 – Core (WS-Addressing), 9 May 2006, http://www.w3.org/TR/2006/REC-ws-addr-core-20060509

[DIME] http://www.gotdotnet.com/team/xml_wsspecs/dime/draft-nielsen-dime-01.txt

[MTOM] http://www.w3.org/TR/soap12-mtom

[SOAP 1.1] http://www.w3.org/TR/soap11

[Base64] http://rfc.net/rfc3548.html

Appendix A: DIME Transfer Mechanism (http://schemas.ggf.org/byteio/2005/10/transfer-mechanisms/dime)

As with "profiles" for any transfer mechanism, the information about the DIME transfer mechanism is meant to profile ONLY the xsd:any element contained in the byteio:transfer-information-type schema. All other defined content for this structure MUST remain intact and unaltered. Note that for messages which do not carry a bulk data payload (the response from a write, or the request to a read operation), this element will be empty.

Due to the nature of the DIME transfer protocol [**DIME**], use of this transfer mechanism does not require any additional information to be included with the byteio:transfer-information-type element. As such, the profile for http://schemas.ggf.org/byteio/2005/10/transfer-mechanisms/dime is empty.

The following is the XML schema for the DIME transfer mechanism profile of the byteio:transfer-information-type element:

```
<byteio:transfer-information-type
        transfer-mechanism="http://schemas.ggf.org/byteio/2005/10/transfer-mechanisms/dime">
</byteio:transfer-information-type>
```

The components of the transfer-information-type data type are unchanged in content from the un-profiled version of this structure.

Appendix B: MTOM Transfer Mechanism (http://schemas.ggf.org/byteio/2005/10/transfer-mechanisms/mtom)

As with "profiles" for any transfer mechanism, the information about the MTOM transfer mechanism is meant to profile ONLY the xsd:any element contained in the byteio:transfer-information-type schema. All other defined content for this structure MUST remain intact and unaltered. Note that for messages which do not carry a bulk data payload (the response from a write, or the request to a read operation), this element will be empty.

Due to the nature of the MTOM transfer protocol [**MTOM**], use of this transfer mechanism does not require any additional information to be included with the byteio:transfer-information-type element. As such, the profile for http://schemas.ggf.org/byteio/2005/10/transfer-mechanisms/mtom is empty.

The following is the XML schema for the MTOM transfer mechanism profile of the byteio:transfer-information-type element:

```
<byteio:transfer-information-type
        transfer-mechanism="http://schemas.ggf.org/byteio/2005/10/transfer-mechanisms/mtom">
</byteio:transfer-information-type>
```

The components of the transfer-information-type data type are unchanged in content from the un-profiled version of this structure.

Appendix C: Simple Transfer Mechanism (http://schemas.ggf.org/byteio/2005/10/transfer-mechanisms/simple)

As with "profiles" for any transfer mechanism, the information about the simple transfer mechanism is meant to profile ONLY the xsd:any element contained in the byteio:transfer-information-type schema. All other defined content for this structure MUST remain intact and unaltered. Note that for messages which do not carry a bulk data payload (the response from a write, or the request to a read operation), this element will be empty.

The following is the XML schema for the simple transfer mechanism profile of the byteio:transfer-information-type element:

```
<byteio:transfer-information-type
        transfer-mechanism="http://schemas.ggf.org/byteio/2005/10/transfer-mechanisms/simple">
        <byteio:data> xsd:base64Binary </byteio:data>*
</byteio:transfer-information-type>
```

The components of the transfer-information-type data type are unchanged in content from the un-profiled version of this structure with the addition of the following component:

/byteio:transfer-information-type/byteio:data

The Base64 [**Base64**] encoded block of data being transferred. This element is not present when no actual bulk data is being transferred (for example, as the result message from a write operation.