Network Working Group
Internet-Draft
Expires: March 15, 2004

A. Romanov
Juniper Networks

September 15, 2003

Some Considerations for SNMP Agent Developers
draft-aromanov-snmp-hiqa-06

Status of this Memo

Copyright Notice

Abstract

SNMP is a ubiquitous protocol. Many software developers working in the embedded space develop or interface with MIB handlers and SNMP agents one way or another. Often these developers are unfamiliar with SNMP standards and overlook a number of subtle points. This document will provide a list of steps and rules to avoid common problems in order to develop a high quality SNMP agent.

IESG Note

This document represents the opinions of the author. It has not been widely reviewed in the IETF. Publication of this document does not mean endorsement by the IESG or the IETF (or SNMP) community.

Table of Contents

1. Overview

The goal of this memo is to facilitate development of SNMP agents in the context of SNMP agent frameworks. Modern SNMP agent frameworks are mature and they provide a good base to build a high-quality agent. These frameworks relieve an application developer from the bulk of the work related to the protocol transaction handling. However, there are still issues that have to be taken care of by an application developer. Unfortunately, there is widespread misunderstanding of some of the fine issues in this area. Moreover, there are new companies entering SNMP framework business, companies that develop their own frameworks and numerous companies that do deep modifications of existing frameworks. Often, even the most experienced developers miss or disregard one or more of the issues addressed in this memo.

2. Index Processing Issues

An SNMP instance is identified by an Object Identifier representing the object name appended with a sequence of sub-identifiers (sub-IDs) representing the index of the instance (we will call it an `index sequence') [RFC2578]. In most cases it is left to the MIB developer to find a variable instance matching an index sequence.

Often it is done in a simple straightforward way: for every row in the table the agent converts values of index variables into index sequence and then chooses the best matching entry, if any. The advantage of this method is that it avoids most if not all of the index processing pitfalls discussed below. However, there are performance and functionality trade-offs associated with this method. So, most implementors choose another method, where the index sequence is first converted into a set of index variables and then these values are used as a search criterion in the instance database. The rest of this section is devoted to a discussion of issues associated with the latter method.

2.1 Index Processing for Get and Set Requests

Fortunately, few problems arise in this area. The first thing to do is to check the length of the index sequence and if it is inconsistent with the required length, `noSuchInstance' (`noCreation' if it is Set request) must be a result of the operation. If `IMPLIED' keyword is present in the INDEX clause then a string and and object identifier of size N are encoded as N sub-IDs. When this key word is not present, the length of the string/object identifier is encoded as the first sub-ID of the index sequence. We will call such string- and object identifier-valued indexes 'non-IMPLIED' throughout this document. The value sub-ID representing the length

of string/object identifier must be checked against overall length of the index sequence too.

Second, the range of an sub-ID (0-4294967295) may be wider than the range of the variable it is being mapped into. For example, the range of integers used for index components is 0-2147483647, while the range of IP address components and the range of string components is 0-255. So, sub-ID 4123456789 indicates a non-existent integer indexed instance, index sequence 1.2.345.4 indicates a non-existent instance indexed by an IP address or IP mask and index sequence 4.65.66.670.68 indicates a non-existent instance indexed by a non-IMPLIED string. In all cases of incorrect range `noSuchInstance' (`noCreation') must be the result of the operation.

So, index processing for Get and Set Requests is simple: check the length of the index sequence, check the range of every sub-ID, and in case of any problems return `noSuchInstance' (`noCreation').

2.2 Index Processing for GetNext and GetBulk Requests

A properly implemented SNMP agent does not assume that a Network Management Station (NMS) will necessarily provide an index sequence in a GetNextRequest-PDU or GetBulkRequest-PDU that specifies an actual or potential object instance. For example, when using the TCP MIB to find the first remote host connected to a particular local TCP port, an NMS application might submit a GetNextRequest-PDU with a partial index containing only the local address and local port. It seems reasonable to start from general principles and then apply them to the particular cases of various index specifications.

1. If the index sequence is longer than a properly formed one, it must be truncated. For example, if MIB variable is indexed by an IP address, then the first instance after 1.2.3.4 and the first instance after 1.2.3.4.5.6.7.8 are the same instance.

2. If the index sequence is shorter than the length of a properly formed one,

   (a) pad the index sequence with zeros and then

   (b) check whether an instance exists that exactly matched the padded index sequence. For example, if the MIB variable is indexed by an IP address, then the first instance after 1.2.3 is 1.2.3.0 if such instance exists. Skipping step (b) is a very popular bug.

3. Perform sub-ID range checking, it must start at the end of index sequence and progress towards its beginning. If the supplied

index sequence contains a sub-ID, which is greater than the limit imposed by underlying application variable, then

(a) if this sub-ID is the first one in the index sequence advance to the next object in the MIB view; otherwise

(b) if previous sub-ID is greater or equal to the limit move to the previous sub-ID and start processing from the step (a); otherwise

(c) increment the previous sub-ID, truncate index sequence starting from the current sub-ID and then go to step 2 above.

Appendix A contains index processing examples for the most popular cases.

3. Issues Related to Set-Request Processing

3.1 Consistency Checking

Unfortunately, there is a lot of confusion in the developer community with regard to the practical requirements of the depth and sophistication of consistency checking. Some developers assume that the standard requires that an agent should be able to verify consistency of every combination of variables that would fit into biggest SetRequest-PDU. Naturally, they feel that this is an absolutely unrealistic requirement and they resort to completely ignoring it. Others simply do best effort consistency checking, with the actual meaning of 'best effort' varying markedly from product to product and even from MIB to MIB within the same product. Some companies build their own agent frameworks that impose severe restrictions on the ability of an agent to do effective consistency checking and some companies build agent frameworks that waste resources providing capabilities far beyond practical necessity. In many cases an agent fails to complain if it receives a SetRequest-PDU that is more complicated than it is designed to process.

Actually, the standard simply requires that

(a) agent check consistency of every variable in the PDU vs. the current managed device status and other variables in the PDU;

(b) if agent is unable to determine consistency (e.g., if the PDU has too many variables for a particular agent implementation to analyze) then 'genErr' should be returned. [RFC3416].

The actual requirements on consistency checking abilities imposed by the standard are left to the developer, i.e., as in many other cases,

the standard relies on the marketplace instead of specifying precise level. For example if a developer aims too low, there will be problems with managing a device in the field and hence a considerable marketplace pressure to rectify the situation; and if a developer aims too high, it will negatively affect time to market and development costs.

In practical terms, the standards do not allow implementing an SNMP agent accepting only one variable per SetRequest-PDU. It is not explicitly prohibited by protocol operations [RFC3416], however, all SNMP agents have to implement the SNMPv2-MIB [RFC3418]. This MIB contains a TestAndIncr [RFC2579] variable snmpSetSerialNo. TestAndIncr objects (often called spin-locks) are intended to control access to other objects, so they have to be present and processed in the PDU together with the variables that they control access to.

To avoid further confusion, it seems reasonable to explicitly spell out the requirement for the "minimal" implementation of an SNMP agent:

(a) an agent must be able to properly check consistency of the following combination of variables (regardless of their order in the PDU) at least: (1) snmpSetSerialNo, (2) any variable, and (3) any combination of spin-lock variables associated with the above variable, if any;

(b) an agent must return `genErr' if the complexity of the SetRequest-PDU exceeds the agent's ability to perform consistency checking: e.g. if the PDU contains any other variable. If an agent is not able to check consistency of a full row in the conceptual table it should use`createAndWait' method of row creation.

A minimal implementation, though valid, is very limiting in many practical cases. The market place is the ultimate judge, but in most practical cases the "reasonable" implementation of an SNMP agent will suffice. Such an implementation should support row creation with `createAndGo' and it should provide consistency checking extended at least to the variables belonging to a single row in the conceptual table. This reasonable implementation provides substantial additional benefits, with minimal efforts comparing to minimal implementation. This level is well supported by practically all available agent frameworks.

Nothing prevents a developer from exceeding this reasonable implementation level. Let us call such implementations "advanced". Also, it is perfectly legal to mix various levels of implementations within the same agent.

Organizations developing or customizing SNMP agent frameworks have to be very careful to select an appropriate maximum implementation level to be supported by the framework. For example, if a framework supports only a minimal implementation, it will be hardly possible to implement legacy MIBs with tables without RowStatus component.

Also, there is an often-overlooked issue mostly related to the consistency checking in advanced implementations. There are always a number of managed system parameters where consistency checking, resource allocation and/or undo operations are practically impossible to accomplish with 100% level of reliability. Fortunately, as a rule these operations are inherently atomic and the failure does not change the management system state. Consistency checks for these cases should not allow these variables to be mixed with any other non spin-lock variables, so the dangerous operation would rely on inherent atomicity instead of checking.

3.2 Miscellaneous Set Request Issues

The intended use of `createAndWait' and `notInService' RowStatus values is to create and manipulate very long rows. Otherwise, they do not provide any additional value, so reasonable and advanced implementations of an SNMP agent may choose not to support these values for MIBs with rows of normal length. Naturally, a minimal implementation must support 'createAndWait'.

An SNMP agent should not ever find itself in the situation where it will return `undoFailed'.

4. Agent Design Issues

There are a number of design issues to be considered. It may require a separate memo to discuss each of them in detail. This memo will be limited to a brief listing of often overlooked items.

1. The spectrum and frequency of requests issued by NMSs are unpredictable and there is always the possibility of NMS bugs, which can result in excessive load on the SNMP agent. It is essential to run SNMP agents as a low priority thread or to take other steps to prevent SNMP agent activities from affecting managed system performance. This is also a major security issue, see below.

2. There is a popular design that links rows in the GetNext order and also puts them into a hash table to provide fast access to the current row. It works perfectly well for Get and Set operations and it also works fine for many GetNext cases, when the index sequence exactly matches an existing row. However, an

NMS is under no obligation to provide index of an existing instance as an index sequence, so in some cases a long linear search is unavoidable. So it is important to take some precautions to guarantee that long linear searches will not impact managed system performance (e.g., along the lines of item (1) above).

3. On systems with memory protection, it is advisable to map tables into read-only shared memory, because user space-kernel space transitions are very expensive. Again along the lines of the item (1) above, kernel transactions should be limited only to the area where it is absolutely essential: namely Set requests.

4. Often, it is desirable to provide a common backend for various management interfaces (SNMP, WEB, CORBA, CLI, etc.). It is surprisingly popular to select an SNMP agent as such a backend. Experience shows that in reality it is a very poor choice of management system design, unless the managed device is a truly trivial one.

5. Intellectual Property

The IETF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on the IETF's procedures with respect to rights in standards-track and standards-related documentation can be found in [RFC2028]. Copies 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 implementors or users of this specification can be obtained from the IETF Secretariat.

The IETF 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 standard. Please address the information to the IETF Executive Director.

6. Security Considerations

SNMPv3 security specifically does not protect against denial of service attacks [RFC3414], so SNMPv3 entities are relatively vulnerable to these attacks: in most configurations NMSs make a

substantial use of insecure communications to convey essential information, agent allows pretty significant replay window, which could be exploited to overload the managed system with requests. Using complex instance level granularity access greatly aggravates the situation.

The author recommends to strictly follow recommendation to implement SNMP as low priority thread in order to eliminate vulnerabilities associated with the denial of service attacks exploiting replay windows. For the same purpose the author recommends that an agent start any Set request with processing of the snmpSetSerialNo if it is present in the PDU. Although not related to the agent side, it is important to remember that every NMS issuing a Set request without snmpSetSerialNo exposes an agent to a possible denial of service attack.

Also, SNMPv3 agent security configuration is a complex matter, even minor imperfection in the agent's security configuration may expose the managed system to the inappropriate level of the risk.

The author recommends to have a built-in possibility to start an agent in `high-security mode' where it will drop all insecure communications delivered to it and will never emit an insecure communication on its own, regardless of its configuration parameters.

References

[RFC2578] McCloghrie, K., Perkins, D., Schoenwaelder, J., Case, J., McCloghrie, K., Rose, M. and S. Waldbusser, "Structure of Management Information Version 2 (SMIv2)", STD 58, RFC 2578, April 1999.

[RFC3416] Presuhn, R., "Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP)", STD 62, RFC 3416, December 2002.

[RFC3418] Presuhn, R., "Management Information Base (MIB) for the Simple Network Management Protocol (SNMP)", STD 62, RFC 3418, December 2002.

[RFC2579] McCloghrie, K., Perkins, D., Schoenwaelder, J., Case, J., McCloghrie, K., Rose, M. and S. Waldbusser, "Textual Conventions for SMIv2", STD 58, RFC 2579, April 1999.

[RFC2028] Hovey, R. and S. Bradner, "The Organizations Involved in the IETF Standards Process", BCP 11, RFC 2028, October 1996.

[RFC3414] Blumenthal, U. and B. Wijnen, "User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3)", STD 62, RFC 3414, December 2002.

[RFC2012] McCloghrie, K., "SNMPv2 Management Information Base for the Transmission Control Protocol using SMIv2", RFC 2012, November 1996.

Author's Address

Aleksey Romanov
Juniper Networks, Inc.
10 Technology Park Drive
Westford, MA 01886
US

EMail: aromanov@juniper.net

Appendix A. GetNext and GetBulk Request Index Processing Examples

A.1 Processing Integer Index

Below is a function that converts a part of an index sequence into an integer. This function converts the sub-ID located at offset `off' in a fully formed index sequence, an index sequence supplied by the NMS is represented by `indexSequence' and `indexSequenceLength'. Note that `off' could be greater that or equal to `indexSequenceLength'. If the fully formed index sequence does not end with the integer in question (i.e., contains other index components beyond it), it is quite possible that processing of the next index component will require that the current sub-IDs be incremented; in that case `inBump' will be set to a non-zero value. The maximum acceptable value is passed as `maxIntVal'. The converted integer will be placed into `intVal'. If it is necessary to probe for an exactly matching instance before the converted value can be used, `checkExact' will be set to a non-zero value. This function returns a non-zero value if the previous sub-ID (i.e., the prefix of the index being converted) has to be incremented.

```
int
nextprocSubid2Int(const uint32 *indexSequence,
                  int indexSequenceLength,
                  int off, int inBump,
                  int32 maxIntVal, int32 *intVal,
                  int *checkExact)
{
```

```
    uint32 subidVal;

    assert(indexSequence != NULL ||
        indexSequenceLength == 0);
    assert(indexSequenceLength >= 0);
    assert(off >= 0);
    assert(indexSequenceLength > (off+1) || !inBump);
    assert(maxIntVal >= 0);
    assert(intVal != NULL);
    assert(checkExact != NULL);

    if(off >= indexSequenceLength)
      {
        /* Index sequence is short */
        assert(inBump == 0);
        *intVal = 0;
        *checkExact = 1;
        return 0;
      }

    subidVal = indexSequence[off];
    if(subidVal > (uint32)maxIntVal ||
       (inBump && subidVal == (uint32)maxIntVal))
      {
        /* Sub-ID is out of range */
        *intVal = 0;
        *checkExact = 1;

        return 1;
      }

    if(inBump)
      {
        *intVal = subidVal + 1;
      }
    else
      {
        *intVal = subidVal;
      }

    *checkExact = 0;
    return 0;
  }
```

A.2 Processing IP Address Index

Below is a function that converts a part of an index sequence into an IP address. This function converts sub-IDs starting at offset `off' in a fully formed index sequence, an index sequence supplied by the NMS is represented by `indexSequence' and `indexSequenceLength'. Note that `off' could be greater than or equal to 'indexSequenceLength'. If the fully formed index sequence does not end with the IP address in question (i.e., contains other index components beyond it), it is quite possible that processing of the next index component will require to increment the last sub-ID representing the IP address; in that case `inBump' will be non-zero. The converted IP address (in host order) will be placed into `addrVal'. If it is necessary to probe for an exactly matching instance before the converted value can be used, `checkExact' will be set to a non-zero value. This function returns a non-zero value if the previous sub-IDs (i.e., the prefix of the index being converted) has to be incremented.

```
int
nextprocSubid2IpAddr(const uint32 *indexSequence,
                     int indexSequenceLength,
                     int off, int inBump,
                     uint32 *addrVal,
                     int *checkExact)
{
  const uint32 *subid, *first, *last;
  uint32 tmp;
  int exact;

  assert(indexSequence != NULL ||
      indexSequenceLength == 0);
  assert(indexSequenceLength >= 0);
  assert(off >= 0);
  assert(indexSequenceLength > (off + 5) || !inBump);
  assert(addrVal != NULL);
  assert(checkExact != NULL);

  if(off >= indexSequenceLength)
    {
      /* Index sequence is short */
      assert(inBump == 0);
      *addrVal = 0;
      *checkExact = 1;
      return 0;
    }
```

```
     first = &indexSequence[off];

     exact = 0;

     if(indexSequenceLength >= (off + 4))
       {
         /* We have full address specified */
         last = &indexSequence[off+3];
       }
     else
       {
         assert(!inBump);
         last = &indexSequence[indexSequenceLength-1];
         exact = 1;
       }

     tmp = 0;

     for(subid=last; subid>=first; subid--)
       {
         if(*subid > 255 || (inBump && *subid == 255))
        {
           if(subid == first)
             {
               *addrVal = 0;
               *checkExact = 1;
               return 1;
             }

           tmp = 0;
           exact = 1;
           inBump = 1;

           continue;
        }

         if(inBump)
        {
           tmp += (((*subid) + 1) <<
                   8*(3 - (subid - first)));
           inBump = 0;
        }
         else
        {
           tmp += ((*subid) <<
                   8*(3 - (subid - first)));
        }
       }
```

```
  assert(!inBump);

  *addrVal = tmp;
  *checkExact = exact;

  return 0;
}
```

A.3 Processing Non-IMPLIED String Index

The first element of this index is the length of the string, so "bb" would go before "aaa", which may be counterintuitive for developers accustomed to lexicographic string ordering.

If the non-IMPLIED string is not the last component of an index, a program has to perform an additional step, in order to determine presence and location of the next component in an index sequence. The function below checks sub-IDs located at offset `off' in a fully formed index sequence, an index sequence supplied by the NMS is represented by `indexSequence' and `indexSequenceLength'. Note that `off' could be greater than or equal to `indexSequenceLength'. The limit on the string length is passed as `maxStringLength'. If the next element is present in the index sequence this function will return a non-zero value and the its offset will be passed in `nextVarOff'.

```
int
nextprocSubid2StrCheck(const uint32 *indexSequence,
                    int indexSequenceLength,
                    int off, int maxStringLength,
                    int *nextVarOff)
{
  assert(indexSequence != NULL ||
      indexSequenceLength == 0);
  assert(indexSequenceLength >= 0);
  assert(off >= 0);
  assert(maxStringLength > 0);
  assert(nextVarOff != NULL);

  if(off >= indexSequenceLength)
    {
      /* There is not enough sub-IDs even for the
      string not speaking about next value */
      return 0;
    }
```

```
    if(maxStringLength > 128)
      {
        /* There is no point to deal with
        strings longer than the whole name
        length limit imposed by protocol */
        maxStringLength = 128;
      }

    if(indexSequence[off] > maxStringLength)
      {
        /* We will have to bump anyway so
        the presence or absence of the next
        component is irrelevant */
        return 0;
      }

    /* Next component has to be checked */
    *nextVarOff = off + 1 + indexSequence[off];
    return 1;
  }
```

Below is a function that converts a part of an index sequence into an array of unsigned characters. This function converts sub-IDs located at offset `off' in a fully formed index sequence, an index sequence supplied by the NMS is represented by `indexSequence' and `indexSequenceLength'. Note that `off' could be greater than or equal to `indexSequenceLength'. If the fully formed index sequence does not end with the sequence in question (i.e., contains other index components beyond it), it is quite possible that processing of the next index component will require to increment the last sub-IDs representing the string; in that case `inBump' will be non-zero. The converted string will be placed into `stringVal', the length of available buffer is passed as `maxStringLength', and the length of processed string is placed into `stringLength'. If it is necessary to probe for an exactly matching instance before the converted value can be used, `checkExact' will be set to a non-zero value. This function returns a non-zero value if the previous sub-ID (i.e., the prefix of the index being converted) has to be incremented.

```
int
nextprocSubid2Str(const uint32 *indexSequence,
                  int indexSequenceLength,
                  int off, int inBump,
                  int maxStringLength,
                  int *stringLength,
                  uint8* stringVal,
                  int *checkExact)
```

```
   {
     const uint32 *subid, *first, *last;
     int len;
     uint8 *s, *resetStart;

     assert(indexSequence != NULL ||
         indexSequenceLength == 0);
     assert(indexSequenceLength >= 0);
     assert(off >= 0);
     assert(maxStringLength >= 0);
     assert(stringLength != NULL);
     assert(stringVal != NULL);
     assert(checkExact != NULL);

     if(off >= indexSequenceLength)
       {
         /* Index sequence is short */
         *stringLength = 0;
         *checkExact = 1;
         return 0;
       }

     if(maxStringLength > 128)
       {
         /* There is no point to deal with
         strings longer than the whole name
         length limit imposed by protocol */
         maxStringLength = 128;
       }

     len = (int)indexSequence[off];

     if(inBump && len == 0)
       {
         if(maxStringLength == 0)
        {
           *stringLength = 0;
           *checkExact = 1;
           return 1;
        }

         *stringLength = 1;
         stringVal[0] = 0;
         *checkExact = 1;
         return 0;
       }

     if(len == 0)
```

```
      {
        /* Empty string */
        *stringLength = 0;
        *checkExact = 0;
        return 0;
      }

    if(len > (uint32)maxStringLength)
      {
        /* Length component indicates length which is too big */
        *stringLength = 0;
        *checkExact = 1;
        return 1;
      }

    off++;

    if(off == indexSequenceLength)
      {
        /* Only length is present */
        memset(stringVal, 0, len);
        *stringLength = len;
        *checkExact = 1;
        return 0;
      }

    first = &indexSequence[off];

    if(indexSequenceLength >= (off + len))
      {
        /* We have full string provided */
        last = &indexSequence[off+len-1];
        resetStart = NULL;
      }
    else
      {
        /* Not a full string */
        assert(inBump == 0);
        last = &indexSequence[indexSequenceLength-1];
        resetStart = stringVal + (indexSequenceLength
                                  - off);
      }

    for(subid=last,s=stringVal+(last - first);
        subid>=first; subid--,s--)
      {
        assert((s - stringVal) == (subid - first));
```

```
         if(*subid > 255 || (inBump && *subid == 255))
        {
          resetStart = s;
          inBump = 1;

          continue;
        }

         if(inBump)
        {
          *s = (uint8) ((*subid) + 1);
          inBump = 0;
        }
         else
        {
          *s = (uint8)(*subid);
        }
       }

     if(inBump)
       {
         if(len == maxStringLength)
        {
          *stringLength = 0;
          *checkExact = 1;
          return 1;
        }

         len++;

         memset(stringVal, 0, len);

         *stringLength = len;
         *checkExact = 1;
         return 0;
       }

     *stringLength = len;

     if(resetStart != NULL)
       {
         assert((resetStart - stringVal) < len);
         memset(resetStart, 0,
             (len - (resetStart - stringVal)));
         *checkExact = 1;
       }
     else
       {
```

```
        *checkExact = 0;
      }

    return 0;
  }
```

A.4 Putting It All Together

Consider an example of tcpConnTable, it is indexed by tcpConnLocalAddress, tcpConnLocalPort, tcpConnRemAddress and tcpConnRemPort where the corresponding index sequence offsets are 0, 4, 6, and 10 [RFC2012]

```
int
nextTcpTableEntry(const uint32 *indexSequence,
                  int indexSequenceLength,
                  struct tcpTableEntry *e)
{
  int ret, bump, exact, curExact;
  int32 localPort, remotePort;
  uint32 localAddr, remoteAddr;

  exact = 0;

  bump = nextprocSubid2Int(indexSequence,
                           indexSequenceLength, 10,
                           0, 0xffff, &remotePort,
                           &curExact);

  if(curExact)
    {
      exact = 1;
    }

  bump = nextprocSubid2IpAddr(indexSequence,
                              indexSequenceLength, 6,
                              bump, &remoteAddr,
                              &curExact);
  if(curExact)
    {
      exact = 1;
      remotePort = 0;
    }

  bump = nextprocSubid2Int(indexSequence,
                           indexSequenceLength, 4,
```

```
                          bump, 0xffff,
                          &localPort, &curExact);
    if(curExact)
      {
        exact = 1;
        remotePort = 0;
        remoteAddr = 0;
      }

    bump = nextprocSubid2IpAddr(indexSequence,
                            indexSequenceLength, 0,
                            bump, &localAddr,
                            &curExact);
    if(bump)
      {
        return NOTFOUND;
      }

    if(curExact)
      {
        exact = 1;
        remotePort = 0;
        remoteAddr = 0;
        localPort  = 0;
      }

    ret = NOTFOUND;

    if(exact)
      {
        ret = retrieveTcpConnection(localAddr,
                                localPort,
                                remoteAddr,
                                remotePort, e);
      }

    if(ret == NOTFOUND)
      {
        ret = retrieveNextTcpConnection(localAddr,
                                    localPort,
                                    remoteAddr,
                                    remotePort,
                                    e);
      }

    return ret;
  }
```

Appendix B. Acknowledgments

Author is grateful to C.M.Heard and J.Perreault for thoughtful review and essential improvements incorporated in this memo. Author is deeply thankful to M.Rose for XML specifications and tools used to write this memo.

Full Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.