pcap-file.cc

Go to the documentation of this file.

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2009 University of Washington
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 * Author:  Craig Dowell (craigdo@ee.washington.edu)
 */

#include <iostream>
#include <cstring>
#include "ns3/assert.h"
#include "ns3/packet.h"
#include "ns3/fatal-error.h"
#include "ns3/fatal-impl.h"
#include "ns3/header.h"
#include "ns3/buffer.h"
#include "pcap-file.h"
#include "ns3/log.h"
#include "ns3/build-profile.h"
//
// This file is used as part of the ns-3 test framework, so please refrain from
// adding any ns-3 specific constructs such as Packet to this file.
//

namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("PcapFile");

const uint32_t MAGIC = 0xa1b2c3d4;
const uint32_t SWAPPED_MAGIC = 0xd4c3b2a1;
const uint32_t NS_MAGIC = 0xa1b23c4d;
const uint32_t NS_SWAPPED_MAGIC = 0x4d3cb2a1;
const uint16_t VERSION_MAJOR = 2;
const uint16_t VERSION_MINOR = 4;
PcapFile::PcapFile ()
  : m_file (),
    m_swapMode (false),
    m_nanosecMode (false)
{
  NS_LOG_FUNCTION (this);
  FatalImpl::RegisterStream (&m_file);
}

PcapFile::~PcapFile ()
{
  NS_LOG_FUNCTION (this);
  FatalImpl::UnregisterStream (&m_file);
  Close ();
}


bool
PcapFile::Fail (void) const
{
  NS_LOG_FUNCTION (this);
  return m_file.fail ();
}
bool
PcapFile::Eof (void) const
{
  NS_LOG_FUNCTION (this);
  return m_file.eof ();
}
void
PcapFile::Clear (void)
{
  NS_LOG_FUNCTION (this);
  m_file.clear ();
}


```

```
87  void
88  PcapFile::Close (void)
89  {
90    NS_LOG_FUNCTION (this);
91    m_file.close ();
92  }
93
94  uint32_t
95  PcapFile::GetMagic (void)
96  {
97    NS_LOG_FUNCTION (this);
98    return m_fileHeader.m_magicNumber;
99  }
100
101 uint16_t
102 PcapFile::GetVersionMajor (void)
103 {
104   NS_LOG_FUNCTION (this);
105   return m_fileHeader.m_versionMajor;
106 }
107
108 uint16_t
109 PcapFile::GetVersionMinor (void)
110 {
111   NS_LOG_FUNCTION (this);
112   return m_fileHeader.m_versionMinor;
113 }
114
115 int32_t
116 PcapFile::GetTimeZoneOffset (void)
117 {
118   NS_LOG_FUNCTION (this);
119   return m_fileHeader.m_zone;
120 }
121
122 uint32_t
123 PcapFile::GetSigFigs (void)
124 {
125   NS_LOG_FUNCTION (this);
126   return m_fileHeader.m_sigFigs;
127 }
128
129 uint32_t
130 PcapFile::GetSnapLen (void)
131 {
132   NS_LOG_FUNCTION (this);
133   return m_fileHeader.m_snapLen;
134 }
135
136 uint32_t
137 PcapFile::GetDataLinkType (void)
138 {
139   NS_LOG_FUNCTION (this);
140   return m_fileHeader.m_type;
141 }
142
143 bool
144 PcapFile::GetSwapMode (void)
145 {
146   NS_LOG_FUNCTION (this);
147   return m_swapMode;
148 }
149
150 bool
151 PcapFile::IsNanoSecMode (void)
152 {
153   NS_LOG_FUNCTION (this);
154   return m_nanosecMode;
155 }
156
157 uint8_t
158 PcapFile::Swap (uint8_t val)
159 {
160   NS_LOG_FUNCTION (this << static_cast<uint32_t> (val));
161   return val;
162 }
163
164 uint16_t
165 PcapFile::Swap (uint16_t val)
166 {
167   NS_LOG_FUNCTION (this << val);
168   return ((val >> 8) & 0x00ff) | ((val << 8) & 0xff00);
169 }
170
171 uint32_t
172 PcapFile::Swap (uint32_t val)
173 {
174   NS_LOG_FUNCTION (this << val);
175   return ((val >> 24) & 0x000000ff) | ((val >> 8) & 0x0000ff00) | ((val << 8) & 0x00ff0000)
    | ((val << 24) & 0xff000000);
176 }
```

```
177
178 void
179 PcapFile::Swap (PcapFileHeader *from, PcapFileHeader *to)
180 {
181   NS_LOG_FUNCTION (this << from << to);
182   to->m_magicNumber = Swap (from->m_magicNumber);
183   to->m_versionMajor = Swap (from->m_versionMajor);
184   to->m_versionMinor = Swap (from->m_versionMinor);
185   to->m_zone = Swap (uint32_t (from->m_zone));
186   to->m_sigFigs = Swap (from->m_sigFigs);
187   to->m_snapLen = Swap (from->m_snapLen);
188   to->m_type = Swap (from->m_type);
189 }
190
191 void
192 PcapFile::Swap (PcapRecordHeader *from, PcapRecordHeader *to)
193 {
194   NS_LOG_FUNCTION (this << from << to);
195   to->m_tsSec = Swap (from->m_tsSec);
196   to->m_tsUsec = Swap (from->m_tsUsec);
197   to->m_inclLen = Swap (from->m_inclLen);
198   to->m_origLen = Swap (from->m_origLen);
199 }
200
201 void
202 PcapFile::WriteFileHeader (void)
203 {
204   NS_LOG_FUNCTION (this);
205   //
206   // If we're initializing the file, we need to write the pcap file header
207   // at the start of the file.
208   //
209   m_file.seekp (0, std::ios::beg);
210
211   //
212   // We have the ability to write out the pcap file header in a foreign endian
213   // format, so we need a temp place to swap on the way out.
214   //
215   PcapFileHeader header;
216
217   //
218   // the pointer headerOut selects either the swapped or non-swapped version of
219   // the pcap file header.
220   //
221   PcapFileHeader *headerOut = 0;
222
223   if (m_swapMode == false)
224     {
225       headerOut = &m_fileHeader;
226     }
227   else
228     {
229       Swap (&m_fileHeader, &header);
230       headerOut = &header;
231     }
232
233   //
234   // Watch out for memory alignment differences between machines, so write
235   // them all individually.
236   //
237   m_file.write ((const char *)&headerOut->m_magicNumber, sizeof(headerOut->m_magicNumber));
238   m_file.write ((const char *)&headerOut->m_versionMajor, sizeof(headerOut->m_versionMajor));
239   m_file.write ((const char *)&headerOut->m_versionMinor, sizeof(headerOut->m_versionMinor));
240   m_file.write ((const char *)&headerOut->m_zone, sizeof(headerOut->m_zone));
241   m_file.write ((const char *)&headerOut->m_sigFigs, sizeof(headerOut->m_sigFigs));
242   m_file.write ((const char *)&headerOut->m_snapLen, sizeof(headerOut->m_snapLen));
243   m_file.write ((const char *)&headerOut->m_type, sizeof(headerOut->m_type));
244 }
245
246 void
247 PcapFile::ReadAndVerifyFileHeader (void)
248 {
249   NS_LOG_FUNCTION (this);
250   //
251   // Pcap file header is always at the start of the file
252   //
253   m_file.seekg (0, std::ios::beg);
254
255   //
256   // Watch out for memory alignment differences between machines, so read
257   // them all individually.
258   //
259   m_file.read ((char *)&m_fileHeader.m_magicNumber, sizeof(m_fileHeader.m_magicNumber));
260   m_file.read ((char *)&m_fileHeader.m_versionMajor, sizeof(m_fileHeader.m_versionMajor));
261   m_file.read ((char *)&m_fileHeader.m_versionMinor, sizeof(m_fileHeader.m_versionMinor));
262   m_file.read ((char *)&m_fileHeader.m_zone, sizeof(m_fileHeader.m_zone));
263   m_file.read ((char *)&m_fileHeader.m_sigFigs, sizeof(m_fileHeader.m_sigFigs));
264   m_file.read ((char *)&m_fileHeader.m_snapLen, sizeof(m_fileHeader.m_snapLen));
265   m_file.read ((char *)&m_fileHeader.m_type, sizeof(m_fileHeader.m_type));
266
267   if (m_file.fail ())
```

```
268       {
269         return;
270       }
271 
272     //
273     // There are four possible magic numbers that can be there.  Normal and byte
274     // swapped versions of the standard magic number, and normal and byte swapped
275     // versions of the magic number indicating nanosecond resolution timestamps.
276     //
277     if (m_fileHeader.m_magicNumber != MAGIC && m_fileHeader.m_magicNumber != SWAPPED_MAGIC &&
278         m_fileHeader.m_magicNumber != NS_MAGIC && m_fileHeader.m_magicNumber != NS_SWAPPED_MAGIC)
279       {
280         m_file.setstate (std::ios::failbit);
281       }
282 
283     //
284     // If the magic number is swapped, then we can assume that everything else we read
285     // is swapped.
286     //
287     m_swapMode = (m_fileHeader.m_magicNumber == SWAPPED_MAGIC
288                   || m_fileHeader.m_magicNumber == NS_SWAPPED_MAGIC) ? true : false;
289 
290     if (m_swapMode)
291       {
292         Swap (&m_fileHeader, &m_fileHeader);
293       }
294 
295     //
296     // Timestamps can either be microsecond or nanosecond
297     //
298     m_nanosecMode = ((m_fileHeader.m_magicNumber == NS_MAGIC) ||
299                      (m_fileHeader.m_magicNumber == NS_SWAPPED_MAGIC)) ? true : false;
300 
301     //
302     // We only deal with one version of the pcap file format.
303     //
304     if (m_fileHeader.m_versionMajor != VERSION_MAJOR || m_fileHeader.m_versionMinor != VERSION_MI
305       {
306         m_file.setstate (std::ios::failbit);
307       }
308 
309     //
310     // A quick test of reasonablness for the time zone offset corresponding to
311     // a real place on the planet.
312     //
313     if (m_fileHeader.m_zone < -12 || m_fileHeader.m_zone > 12)
314       {
315         m_file.setstate (std::ios::failbit);
316       }
317 
318     if (m_file.fail ())
319       {
320         m_file.close ();
321       }
322 }
323 
324 void
325 PcapFile::Open (std::string const &filename, std::ios::openmode mode)
326 {
327   NS_LOG_FUNCTION (this << filename << mode);
328   NS_ASSERT ((mode & std::ios::app) == 0);
329   NS_ASSERT (!m_file.fail ());
330   //
331   // All pcap files are binary files, so we just do this automatically.
332   //
333   mode |= std::ios::binary;
334 
335   m_filename=filename;
336   m_file.open (filename.c_str (), mode);
337   if (mode & std::ios::in)
338     {
339       // will set the fail bit if file header is invalid.
340       ReadAndVerifyFileHeader ();
341     }
342 }
343 
344 void
345 PcapFile::Init (uint32_t dataLinkType, uint32_t snapLen, int32_t timeZoneCorrection, bool
    swapMode, bool nanosecMode)
346 {
347   NS_LOG_FUNCTION (this << dataLinkType << snapLen << timeZoneCorrection << swapMode);
348 
349   //
350   // Initialize the magic number and nanosecond mode flag
351   //
352   m_nanosecMode = nanosecMode;
353   if (nanosecMode)
354     {
355       m_fileHeader.m_magicNumber = NS_MAGIC;
356     }
357   else
```

```
358       {
359         m_fileHeader.m_magicNumber = MAGIC;
360       }
361 
362     //
363     // Initialize remainder of the in-memory file header.
364     //
365     m_fileHeader.m_versionMajor = VERSION_MAJOR;
366     m_fileHeader.m_versionMinor = VERSION_MINOR;
367     m_fileHeader.m_zone = timeZoneCorrection;
368     m_fileHeader.m_sigFigs = 0;
369     m_fileHeader.m_snapLen = snapLen;
370     m_fileHeader.m_type = dataLinkType;
371 
372     //
373     // We use pcap files for regression testing.  We do byte-for-byte comparisons
374     // in those tests to determine pass or fail.  If we allow big endian systems
375     // to write big endian headers, they will end up byte-swapped and the
376     // regression tests will fail.  Until we get rid of the regression tests, we
377     // have to pick an endianness and stick with it.  The precedent is little
378     // endian, so we set swap mode if required to pick little endian.
379     //
380     // We do want to allow a user or test suite to enable swapmode irrespective
381     // of what we decide here, so we allow setting swapmode from formal parameter
382     // as well.
383     //
384     // So, determine the endianness of the running system.
385     //
386     union {
387       uint32_t a;
388       uint8_t  b[4];
389     } u;
390 
391     u.a = 1;
392     bool bigEndian = u.b[3];
393 
394     //
395     // And set swap mode if requested or we are on a big-endian system.
396     //
397     m_swapMode = swapMode | bigEndian;
398 
399     WriteFileHeader ();
400 }
401 
402 uint32_t
403 PcapFile::WritePacketHeader (uint32_t tsSec, uint32_t tsUsec, uint32_t totalLen)
404 {
405     NS_LOG_FUNCTION (this << tsSec << tsUsec << totalLen);
406     NS_ASSERT (m_file.good ());
407 
408     uint32_t inclLen = totalLen > m_fileHeader.m_snapLen ? m_fileHeader.m_snapLen : totalLen;
409 
410     PcapRecordHeader header;
411     header.m_tsSec = tsSec;
412     header.m_tsUsec = tsUsec;
413     header.m_inclLen = inclLen;
414     header.m_origLen = totalLen;
415 
416     if (m_swapMode)
417       {
418         Swap (&header, &header);
419       }
420 
421     //
422     // Watch out for memory alignment differences between machines, so write
423     // them all individually.
424     //
425     m_file.write ((const char *)&header.m_tsSec, sizeof(header.m_tsSec));
426     m_file.write ((const char *)&header.m_tsUsec, sizeof(header.m_tsUsec));
427     m_file.write ((const char *)&header.m_inclLen, sizeof(header.m_inclLen));
428     m_file.write ((const char *)&header.m_origLen, sizeof(header.m_origLen));
429     NS_BUILD_DEBUG(m_file.flush());
430     return inclLen;
431 }
432 
433 void
434 PcapFile::Write (uint32_t tsSec, uint32_t tsUsec, uint8_t const * const data, uint32_t totalLen
435 {
436     NS_LOG_FUNCTION (this << tsSec << tsUsec << &data << totalLen);
437     uint32_t inclLen = WritePacketHeader (tsSec, tsUsec, totalLen);
438     m_file.write ((const char *)data, inclLen);
439     NS_BUILD_DEBUG(m_file.flush());
440 }
441 
442 void
443 PcapFile::Write (uint32_t tsSec, uint32_t tsUsec, Ptr<const Packet> p)
444 {
445     NS_LOG_FUNCTION (this << tsSec << tsUsec << p);
446     uint32_t inclLen = WritePacketHeader (tsSec, tsUsec, p->GetSize ());
447     p->CopyData (&m_file, inclLen);
448     NS_BUILD_DEBUG(m_file.flush());
```

```
449 }
450
451 void
452 PcapFile::Write (uint32_t tsSec, uint32_t tsUsec, const Header &header, Ptr<const Packet> p)
453 {
454   NS_LOG_FUNCTION (this << tsSec << tsUsec << &header << p);
455   uint32_t headerSize = header.GetSerializedSize ();
456   uint32_t totalSize = headerSize + p->GetSize ();
457   uint32_t inclLen = WritePacketHeader (tsSec, tsUsec, totalSize);
458
459   Buffer headerBuffer;
460   headerBuffer.AddAtStart (headerSize);
461   header.Serialize (headerBuffer.Begin ());
462   uint32_t toCopy = std::min (headerSize, inclLen);
463   headerBuffer.CopyData (&m_file, toCopy);
464   inclLen -= toCopy;
465   p->CopyData (&m_file, inclLen);
466 }
467
468 void
469 PcapFile::Read (
470   uint8_t * const data,
471   uint32_t maxBytes,
472   uint32_t &tsSec,
473   uint32_t &tsUsec,
474   uint32_t &inclLen,
475   uint32_t &origLen,
476   uint32_t &readLen)
477 {
478   NS_LOG_FUNCTION (this << &data <<maxBytes << tsSec << tsUsec << inclLen << origLen << readLen
479   NS_ASSERT (m_file.good ());
480
481   PcapRecordHeader header;
482
483   //
484   // Watch out for memory alignment differences between machines, so read
485   // them all individually.
486   //
487   m_file.read ((char *)&header.m_tsSec, sizeof(header.m_tsSec));
488   m_file.read ((char *)&header.m_tsUsec, sizeof(header.m_tsUsec));
489   m_file.read ((char *)&header.m_inclLen, sizeof(header.m_inclLen));
490   m_file.read ((char *)&header.m_origLen, sizeof(header.m_origLen));
491
492   if (m_file.fail ())
493     {
494       return;
495     }
496
497   if (m_swapMode)
498     {
499       Swap (&header, &header);
500     }
501
502   tsSec = header.m_tsSec;
503   tsUsec = header.m_tsUsec;
504   inclLen = header.m_inclLen;
505   origLen = header.m_origLen;
506
507   //
508   // We don't always want to force the client to keep a maximum length buffer
509   // around so we allow her to specify a minimum number of bytes to read.
510   // Usually 64 bytes is enough information to print all of the headers, so
511   // it isn't typically necessary to read all thousand bytes of an echo packet,
512   // for example, to figure out what is going on.
513   //
514   readLen = maxBytes < header.m_inclLen ? maxBytes : header.m_inclLen;
515   m_file.read ((char *)data, readLen);
516
517   //
518   // To keep the file pointer pointed in the right place, however, we always
519   // need to account for the entire packet as stored originally.
520   //
521   if (readLen < header.m_inclLen)
522     {
523       m_file.seekg (header.m_inclLen - readLen, std::ios::cur);
524     }
525 }
526
527 bool
528 PcapFile::Diff (std::string const & f1, std::string const & f2,
529                 uint32_t & sec, uint32_t & usec, uint32_t & packets,
530                 uint32_t snapLen)
531 {
532   NS_LOG_FUNCTION (f1 << f2 << sec << usec << snapLen);
533   PcapFile pcap1, pcap2;
534   pcap1.Open (f1, std::ios::in);
535   pcap2.Open (f2, std::ios::in);
536   bool bad = pcap1.Fail () || pcap2.Fail ();
537   if (bad)
538     {
539       return true;
```

```
540       }
541
542     uint8_t *data1 = new uint8_t [snapLen] ();
543     uint8_t *data2 = new uint8_t [snapLen] ();
544     uint32_t tsSec1 = 0;
545     uint32_t tsSec2 = 0;
546     uint32_t tsUsec1 = 0;
547     uint32_t tsUsec2 = 0;
548     uint32_t inclLen1 = 0;
549     uint32_t inclLen2 = 0;
550     uint32_t origLen1 = 0;
551     uint32_t origLen2 = 0;
552     uint32_t readLen1 = 0;
553     uint32_t readLen2 = 0;
554     bool diff = false;
555
556     while (!pcap1.Eof () && !pcap2.Eof ())
557       {
558         pcap1.Read (data1, snapLen, tsSec1, tsUsec1, inclLen1, origLen1, readLen1);
559         pcap2.Read (data2, snapLen, tsSec2, tsUsec2, inclLen2, origLen2, readLen2);
560
561         bool same = pcap1.Fail () == pcap2.Fail ();
562         if (!same)
563           {
564             diff = true;
565             break;
566           }
567         if (pcap1.Eof ())
568           {
569             break;
570           }
571
572         ++packets;
573
574         if (tsSec1 != tsSec2 || tsUsec1 != tsUsec2)
575           {
576             diff = true; // Next packet timestamps do not match
577             break;
578           }
579
580         if (readLen1 != readLen2)
581           {
582             diff = true; // Packet lengths do not match
583             break;
584           }
585
586         if (std::memcmp (data1, data2, readLen1) != 0)
587           {
588             diff = true; // Packet data do not match
589             break;
590           }
591       }
592     sec = tsSec1;
593     usec = tsUsec1;
594
595     bad = pcap1.Fail () || pcap2.Fail ();
596     bool eof = pcap1.Eof () && pcap2.Eof ();
597     if (bad && !eof)
598       {
599         diff = true;
600       }
601
602     delete[] data1;
603     delete[] data2;
604
605     return diff;
606 }
607
608 } // namespace ns3
```