Notes on Date & Time
Part 1: Date Calculation

2010/08

Shuo Chen

Contents of this serial

- Strategic date calculation
- Time keeping (in UTC)
- Time zone and daylight saving time
  - Choices between local time and UTC time
- Time in a distributed system
  - How leap seconds affect heartbeat protocol
- Source Code
  - http://github.com/chenshuo/recipes/tree/master/datetime/

Part 1: Strategic date calculation

- Date calculation with Julian Day Number
  - Assume dates after 1900 until we talk about history
- Illustrations of magic formulas
- Design a `Date` class
  - Alternative designs
  - Unit tests
- Libraries available
- Brief history of Gregorian calendar

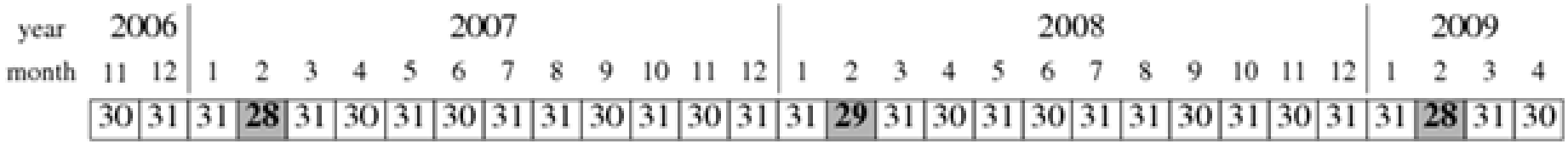
Common date calc tasks

- How many days between two dates
  - 2000-01-01 and 2008-08-08
- Is 2012/01/01 a Sunday?
- When is the 1000th day after 2009/12/25?

- Would be trivial if dates map to consecutive integers and vice versa.

Gregorian calendar

- Days in a month



year	2006		2007												2008												2009			
month	11	12	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4
	30	31	31	**28**	31	30	31	30	31	31	30	31	30	31	31	**29**	31	30	31	30	31	31	30	31	30	31	31	**28**	31	30

- Leap year
  - 97 leap years every 400 years
- A full cycle is 400 years, 400*365+97 = 146097d
- How to number each date with an integer given such an irregular pattern?

Convert Dates to Integers

- Day number of (*year*, *month*, *day*) is sum of
  - Days before Jan 1 of the *year*, def daysYear() as
    - daysYear(year+1) = daysYear(year) + 365 + isLeap(year)
    - daysYear(2004) = daysYear(2003) + 365
    - daysYear(2005) = daysYear(2004) + 366
  - Days in this year before 1st day of this *month*
    - daysMonth(1) = 0, daysMonth(2) = 31,
    - daysMonth(3) = 28 or 29, daysMonth(4) = 31, ...
  - Days in this month, ie. *day*
- Before going on with rolling our own ...

Julian Day Number

- The standard method of representing dates as consecutive positive integers
- Some day predates all record history as day 1
  - 4700+ BC or so
- Integer arithmetic only, without look-up table
- See
  - http://www.faqs.org/faqs/calendars/faq/
  - http://en.wikipedia.org/wiki/Julian_day

Julian Day Number from Date

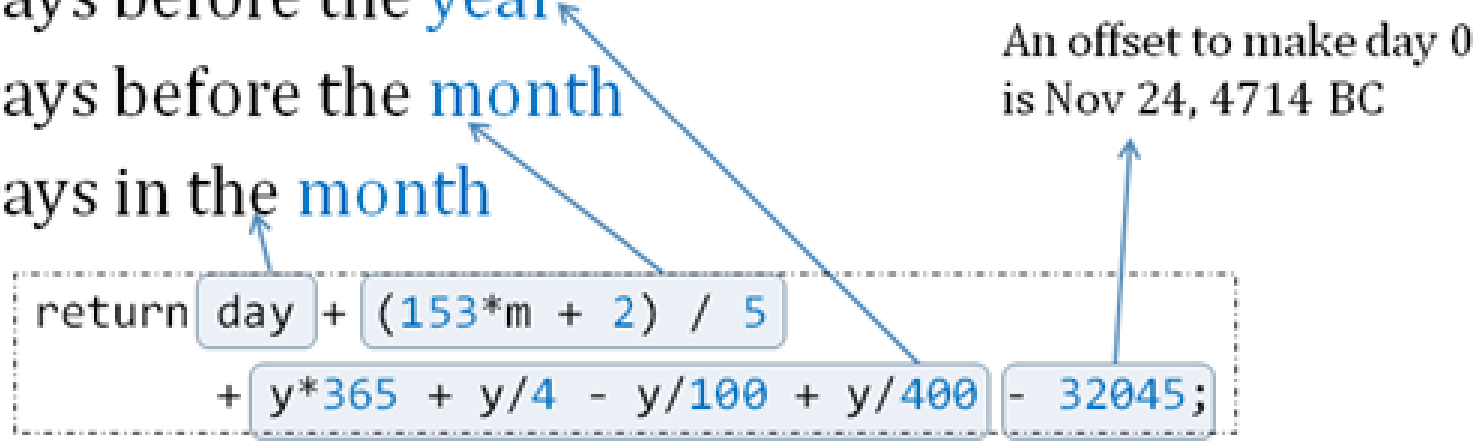
- Code in C, divisions are truncation

```
int toJulianDayNumber(int year, int month, int day)
{
  int a = (14 - month) / 12;
  int y = year + 4800 - a;
  int m = month + 12 * a - 3;
  return day + (153*m + 2) / 5 + y*365
         + y/4 - y/100 + y/400 - 32045;
}
```

- 1 <= month <= 12, 1<= day
- Tons of magic numbers

How does it work?

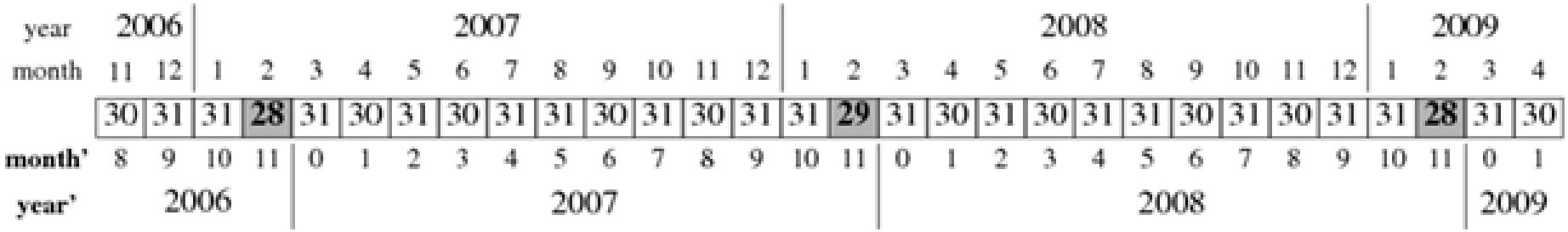
- Given a date (*year*, *month*, *day*), Julian Day Number is sum of
  - Days before the year
  - Days before the month
  - Days in the month



- Why not use year and month directly?
  - *y* and *m* are shifted *year* and *month*

Shift year by two months

- New year starts in March ...



  - March is month 0 in shifted year
  - February is the last month (no. 11) of shifted year
  - Leap day (if any) will be the last day of pervious year
- Length of first 11 months (m=0..10) are fixed
  - 31, 30, 31, 30, 31, 31, 30, 31, 30, 31, 31

Shifting is easy

- First three lines of `toJulianDayNumber()`

```
int a = (14 - month) / 12;  // = (month < 3) ? 1 : 0
int y = year + 4800 - a;
int m = month + 12 * a - 3;
```

  - month = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

$$y = \begin{cases} year + 4800 & month \geq 3 \\ year + 4799 & month < 3 \end{cases}$$

$$m = \begin{cases} month - 3 & month \geq 3 \\ month + 9 & month < 3 \end{cases}$$

- After shifting, much easier to calculate
  - Days before the year, esp. days before the month

Days before the year

- The (shifted) year part of our magic formula

```
y*365 + y/4 - y/100 + y/400
```

  - The last three terms of the expression add in a day for each year that is a leap year.
    - Recall that the leap day counts in pervious year
  - The first term adds a day every 4 years
  - The second subtracts a day back out every 100 years
  - The third adds a day back in every 400 years
- To verify, calc how many leap years between
  - 1~100 (24), 1~200 (48), 1~300 (72), 1~400 (97)

Days before the month

- The most magical part in the formula is about m

```
daysBeforeMonth(m) := (153*m + 2) / 5
```

- It gives cumulative sum of days in month

year	2007												2008												2009	
month	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4	5	6	7	8	9	10	11	12	1	2
days in month	31	**28**	31	30	31	30	31	31	30	31	30	31	31	**29**	31	30	31	30	31	31	30	31	30	31	31	**28**
cum. sum	306	337	0	31	61	92	122	153	184	214	245	275	306	337	0	31	61	92	122	153	184	214	245	275	306	337
m	10	11	0	1	2	3	4	5	6	7	8	9	10	11	0	1	2	3	4	5	6	7	8	9	10	11
y	6806		6807												6808											

- daysBeforeMonth(March) = 0 March is the first month
- daysBeforeMonth(April) = 31 March has 31 days
- daysBeforeMonth(May) = 61 plus April's 30 days
- daysBeforeMonth(Feb) = 337 total days of March to Jan

Days before the month, con't

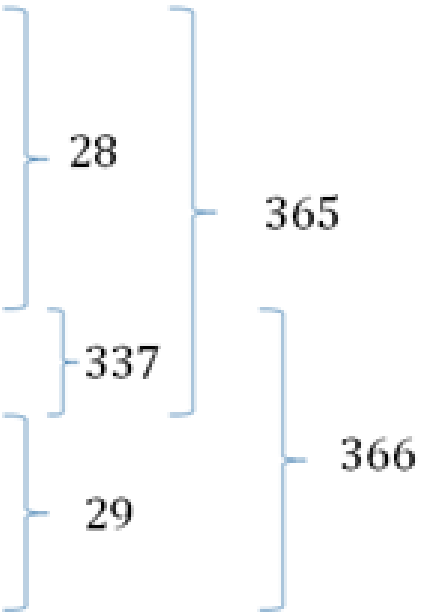
- By shifting 2-month, first 11 months have fixed length, so it's much easier to calc cumulative sum
  - No matter the last month (Feb) has 28 or 29 days, it will be addressed in the days in the month part
- It can be done either with look-up table, built with std::partial_sum(), or a simple function

$$\left\lfloor \frac{153m+2}{5} \right\rfloor$$

- We will see how the coefficients come from
  - Algorithm 199, *Comm. of the ACM*, Aug 1963

Examples

- Calc Julian day number
  - 2007-02-01 2454133
  - 2007-02-28 2454160
  - 2007-02-29* 2454161
  - 2007-03-01 2454161
  - 2008-02-01 2454498
  - 2008-02-29 2454526
  - 2008-03-01 2454527



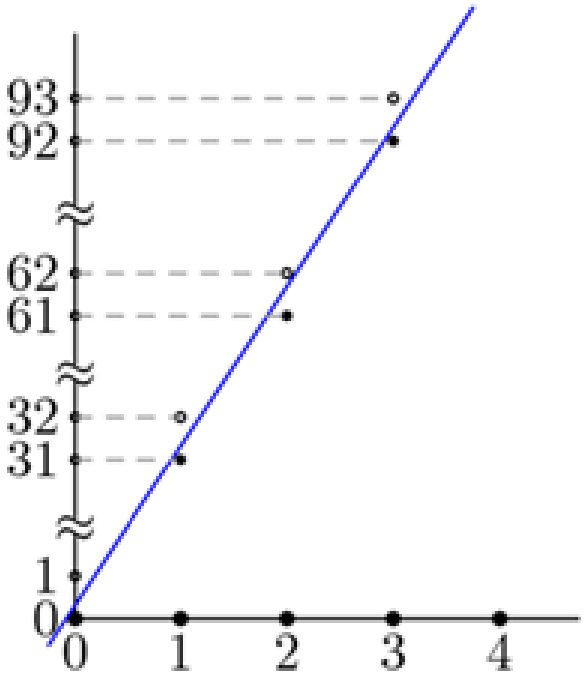
- isLeap(year) := (julianDay(year, 2, 29) != julianDay(year, 3, 1))

Function for days before month

- Find a function $f(m)$, so that

$$
\begin{aligned}
\lfloor f(0) \rfloor = 0 \quad &\Rightarrow \quad 0 \le f(0) < 1 \\
\lfloor f(1) \rfloor = 31 \quad &\Rightarrow \quad 31 \le f(1) < 32 \\
\lfloor f(2) \rfloor = 61 \quad &\Rightarrow \quad 61 \le f(2) < 62 \\
\lfloor f(3) \rfloor = 92 \quad &\Rightarrow \quad 92 \le f(3) < 93 \\
&\vdots \\
\lfloor f(11) \rfloor = 337 \quad &\Rightarrow \quad 337 \le f(11) < 338
\end{aligned}
$$

- That is, find a line drills through each pair of dots

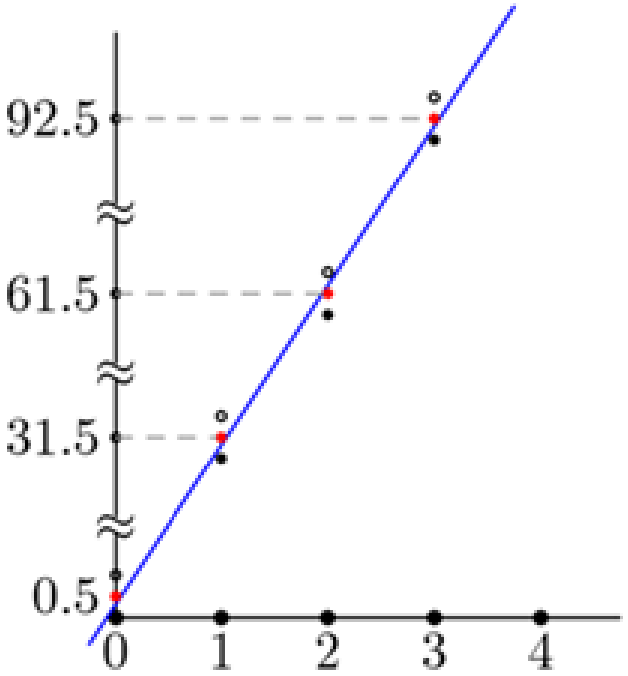


Find a straight line that fits

- Since the number of days of first 11 months are either 30 or 31, we try a straight line first, that is, a linear function

  $y=ax+b$

- To find out a and b, we find a line which is close to the middle points of every pairs
- That is, find a line that fits
  - $x_i = \{0, 1, 2, 3, ..., 11\}$
  - $y_i = \{0.5, 31.5, 61.5, 92.5, ..., 337.5\}$



Simple linear regression

- The estimation of a and b are

$$\hat{a} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2} = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n\sum_{i=1}^{n}(x_i^2) - \left(\sum_{i=1}^{n} x_i\right)^2}$$

$$\hat{b} = \bar{y} - \hat{a}\bar{x}$$

  - http://en.wikipedia.org/wiki/Simple_linear_regression
  - Result a = 30.601, b = 0.52564 (regress.m in code)
- To get integer coefficients, we could round a and b
  - a = 30.6 and b = 0.5 ⇒ (306*x+5)/10
  - a = 30.6 and b = 0.4 ⇒ (153*x+2)/5
  - Both happen to give correct answers, we're done.

Break down Julian Day Number

- Inverse function of `toJulianDayNumber()`

```
YearMonthDay toYearMonthDay(int julianDayNumber)
{
  int a = julianDayNumber + 32044;
  int b = (4*a + 3) / 146097;
  int c = a - ((b * 146097) / 4);
  int d = (4*c + 3) / 1461;
  int e = c - ((1461 * d) / 4);
  int m = (5*e + 2) / 153;
  YearMonthDay ymd;
  ymd.day = e - ((153*m + 2) / 5) + 1;
  ymd.month = m + 3 - 12 * (m / 10);
  ymd.year = b*100 + d - 4800 + m/10;
  return ymd;
}
```

```
struct YearMonthDay
{
  int year;
  int month; // [1..12]
  int day;
};
```

- Could it possibly be correct?

Two nested cycles of calendar

- A cycle consists four phases, first three phases are in the same length, last phase has one more day
- Outter cycle, 4 centuries, 146097 days
  - Phase 0, year 1~100, 24 leap years, 36524 days
  - Phase 1 and 2, same length (one century)
  - Phase 3, year 301~400, 25 leap years, 36525 days
- Inner cycle, 4 years, 1461 days
  - Phase 0, 1st year, 365 days
  - Phase 1 and 2, same length (one year)
  - Phase 3, 4th year, leap year, 366 days

Examples of cycles

- Outer cycle, 2000-03-01~2400-02-29 (146097d)
  - Phase I, 2000-03-01 ~ 2100-02-28 36524d
  - Phase II, 2100-03-01 ~ 2200-02-28 36524d
  - Phase III, 2200-03-01 ~ 2300-02-28 36524d
  - Phase IV, 2300-03-01 ~ 2400-02-29 36525d
- Inner cycle, 2000-03-01~2004-02-29 (1461d)
  - Phase I, 2000-03-01 ~ 2001-02-28 365d
  - Phase II, 2001-03-01 ~ 2002-02-28 365d
  - Phase III, 2002-03-01 ~ 2003-02-28 365d
  - Phase IV, 2003-03-01 ~ 2004-02-29 366d

Determine phases in cycles

- Formulas to break a day x into phases and offsets

$$p = \left\lfloor \frac{4x+3}{4n+1} \right\rfloor \quad b = \left\lfloor \frac{p(4n+1)}{4} \right\rfloor \quad d = x - b$$

  - n is length of phase (365 or 36524), 4n+1 is cycle len.
  - p is phase, 0~3 for cycle 1, 4~7 for cycle 2, and so on
  - b is start day of phase p
  - d is offset of x in phase p

$$\begin{aligned}
p &= 0 & 0 &\le x < n \\
p &= 1 & n &\le x < 2n \\
p &= 2 & 2n &\le x < 3n \\
p &= 3 & 3n &\le x < 4n+1 \\
p &= 4 & 4n+1 &\le x < (4n+1)+n \\
p &= 7 & (4n+1)+3n &\le x < 2(4n+1)
\end{aligned}$$

- Eg. Day 2000 is
  - Phase 5 of inner cycle
  - 174th day of that year

Put them together

- Review the code of toYearMonthDay()

```
int a = julianDayNumber + 32044; // 0-th day is Mar 1, epoch year
int b = (4*a + 3) / 146097;      // century (phase of outer cycle)
int c = a - ((b * 146097) / 4);  // days in the century
int d = (4*c + 3) / 1461;        // year in the century (inner phase)
int e = c - ((1461 * d) / 4);    // days in the year
int m = (5*e + 2) / 153;         // shifted month in the year
YearMonthDay ymd;
ymd.day = e - ((153*m + 2) / 5) + 1;  // day of the month
ymd.month = m + 3 - 12 * (m / 10);    // normal month
ymd.year = b*100 + d - 4800 + m/10;   // normal year
```

- Are you convinced?

Final note

- Last inner cycle may be incomplete, but doesn't matter. Eg. 2100 is not leap year
  - Inner cycle 2096-03-01~2100-02-28 has 1460 days
  - 2100-03-01 belongs to next (inner and outer) cycle
- Moving leap day to the end of year, simplified a lot
  - The ancient Roman calendar started its year on 1st March
- Integer arithmetic can be tricky and useful

What do we have

- A function maps dates in Gregorian calendar to consecutive positive integers (known as Julian Day Number)
- An inverse function breaks down Julian Day Number to year, month and day
- Be convinced those mappings work (I hope so.)
- Let's put it into real program, design a `Date` class

Design a `Date` class

- For civil or business usage
  - Not for historian (dates before 1900)
  - Not for rocket science or astronomy
- Assuming the current Gregorian calendar always was, and always will be, in effect.
  - See the history of adoption of Gregorian calendar at the end of this slides

Define data member(s)

- The text book way

```
class Date
{
 // ...
 private:
  int year_;
  int month_;
  int day_;
};
```

- The professional way

```
class Date
{
 // ...
 private:
  int julianDayNumber_;
};
```

Full code:
http://github.com/chenshuo/recipes/tree/master/datetime/

Define member functions

- Date()
- Date(int y, int m, int d)
- explicit Date(int jdn)
- explicit Date(const struct tm&)
- // default copy-ctor and assignment
- bool valid() const
- int year() const
- int month() const
- int day() const
- int weekDay() const
- struct YearMonthDay yeaMonthDay() const
- string toIsoString() const

Design decisions

- Make it a value type, and make it *immutable*
- Doesn't provide today()
  - The date should be injected to the system, not simply taken from local time or GMT time.
  - Make testing simple. You don't have to wait for years to get a date like 2012-02-29 for testing
- Month ranges in [1, 12]
  - Let `printf("%4d-%02d-%02d", date.year(), date.month(), date.day())` prints intuitive result
  - Same as other date libraries except C's struct tm

Unit Tests


```
int julianDayNumber = 2415021; // 1900-01-01
int weekDay = 1; // Monday
int days[2][12+1] =  {
    { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
    { 0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }};

for (int year = 1900; year < 2500; ++year) {
  for (int month = 1; month <= 12; ++month) {
    for (int day = 1; day <= days[isLeap(year)][month]; ++day) {
    // check Date d(year, month, day) and d2(julianDayNumber)
    ++julianDayNumber;
    weekDay = (weekDay+1) % 7;
    }
  }
}
```

- Inspired by *The Elements of Programming Style* 2/e, p.54

Alternative designs

- Strong type Year, Month and Day
  - The Most Important Design Guideline?,
    by Scott Meyers, IEEE Software, July/August 2004.
- Move toIsoString() outside of class
  - How Non-Member Functions Improve Encapsulation,
    by Scott Meyers, *CUJ* February 2000.
  - Item 18 and 23 of *Effective C++* 3rd ed.
- Provides more features
  - parseIsoString() // yyyy-mm-dd
  - getNextMonday(), etc.

Available libraries

- Boost::date_time
  - Use it if you need "customization and extension"
- Python datetime
  - http://docs.python.org/library/datetime.html
- Ruby date
  - http://ruby-doc.org/stdlib/libdoc/date/rdoc/
- Java Joda Time
  - http://joda-time.sourceforge.net/

POSIX.1 2003

- Doesn't provide date function
- Date time calc based on Unix time, ie.
  - Seconds since 1970-01-01 00:00:00 UTC
  - Exposed to year 2038 problem on 32-bit platform
    - http://code.google.com/p/y2038/
- Will talk more in coming parts, when we talk UTC time and local time

A bit of history

- Calendars are notoriously idiosyncratic.
  - http://en.wikipedia.org/wiki/Gregorian_calendar#History
- Gregorian calendar begins in 1582.
  Not all countries adopted at same time
  - England, United States (ex. Alaska), 1752
    - `$ cal Sep 1752` # try it on Linux
    - 1700 was a leap year in England history
  - Russia, 1918
    - October Revolution happened on 1917-11-07 Gregorian
    - 1900 was a leap year in Russia history

To be continued

- Time keeping in programs
  - Always record happened event in UTC time
- Time zones and daylight saving time
  - Consider keep scheduled event in local time
  - Do not hardcode time zone offset or DST rules
- Time in a distributed system
  - Two timestamps from two hosts are not comparable, unless you know the clock skew of those two hosts
  - NTP isn't good enough for monitoring latency in LAN