Analysis of 23C3 Sputnik data

Tomasz Rybak
tomasz.rybak@post.pl

This article describes attempts to analyse data coming from Sputnik project gathered during 23rd Chaos Communication Congress. The most significant problem was recovering lost sequence identifiers, and this is main subject of article.

1 Sputnik idea

Sputnik is RFID system intended to trace people in small areas, and buildings. Each person is wearing tag that transmits its identifier in regular time intervals to allow to store this persons position at those moments. System was used during previous, 23rd Congress, and during Chaos Communication Camp 2007. Data from Camp has not yet been released, and this article describes analysis performed on data from 23C3.

After releasing data there were few web pages created describing system and data, and trying to analyse it. The main page of project[1] is maintained by creators of Sputnik system. Wiki of OpenBeacon contains page[2] with discussion about released data. Peter Meerwald came with page[3] presenting come analysis of gathered data. Kaners page[4] contains parser of log files, allowing to get information about only particular ID. My page[5] contains programs and results described in this article.

2 Hardware

Ordinary RFID systems are operating in range of few dozens kHz, and use passive tags. Tag does not contain any power source; it is powered by reader during reading process. So without reader it can do nothing. Sputnik uses active tags; they have own battery and transmit data whatever there is reader listening to it or not. Using own battery allows for having high power and thus high range of transmission. Range in buildings is up to the 10m even through dry walls. Concrete walls tend to block signal. Because transmission occurs at 2.4GHz, human body decreases power by about 50%.

Thanks to own battery tag has control over transmission power and can send signals varying in strength. This allows for estimating distance from reader. During 23C3 25 readers were placed in BCC in such a way that in most cases more than one reader saw tag. This, because of possibility of estimating distance from reader, allows for estimation of position of tag.

First readers were large boxes using Power Ethernet to communicate with the server and to power themselves. During Camp Milosz Meriac presented USB reader[6], small device, powered and transmitting data using USB. It acts like terminal, sending data in text format; computer can receive read packets, and send commands to it. Additionally it can also serve as tag, as it have full transmitter on board. Because it is more sophisticated than tag, user has more control over sent RFID packets. It creates /dev/ttyACM* device and sends text in either "ID,Sequence,strength,flags" or "RX: ID,strength,number" format, depending on version of firmware. It can be reprogrammed directly using USB, without any additional hardware.

[1] http://www.openbeacon.org/
[2] http://wiki.openbeacon.org/wiki/Datamining
[3] http://pmeerw.net/23C3_Sputnik/
[4] http://cakelab.org/ kaner/sputnik_01/
[5] http://www.bogomips.w.tkb.pl/sputnik.html
[6] http://wiki.openbeacon.org/wiki/OpenBeacon_USB

3 Data format

Data gathered during 23C3 was made available as both XML and binary files.

XML file

Consisted of "observation" tags with following attributes:

id ID of tag

time

position position of tag; (0, 0, 0) if unknown

direction always (0, 0, 0)

priority always the same value 24

min-distance always 0.0

max-distance always 255.0

observer URL of aggregating station; only one value present in file

observed-object URL of station together with tag ID

XML file contains very small portion of data that was gathered during 23C3. It has only 357974 entries, where full data set is 11.1 million of observations. It does not contain details of readers used to calculate positions of tags. This omission is important, as about 1/3rd of observations has no meaningful position calculated, probably because in those cases there was not enough data to calculate those positions. Also XML file contains data from only few hours for each day of Congress; probably those are hours when server was active. Number of observations during the Congress stored in XML file is shown in Figure 3.

Because of having no sequence numbers and reading stations used to calculate positions, I did not use XML data in analysis.

Data from binary file was more useful for analysis, although it contained errors. Because of error in server software, identifiers of tags were not saved.

Binary format according to source code

0-4 timestamp

5-8 reader station IP

9 size of frame (0x10)

10 protocol (0x17)

11 flags (0x02 — button pressed)

12 strength of signal

12-16 sequence number

17-20 Tag ID

21-24 check sum

Binary format present in file

0-4 timestamp

5-8 reader station IP

9-12 garbage (used by me to write ID)

13-16 garbage, reversed IP of reader station

17 size of frame (0x10)

18 protocol (0x17)

19 flags (0x02 — button pressed)

20 strength of signal

21-24 sequence number

Missing identifiers made analysis almost impossible. Additional problem were 8 bytes in one of files; information published on OpenBeacon mailing list allowed me to removed those unnecessary bytes and to have full data set. Binary data set had 64K repeated readings — observations that were the same as other observations.

4 Database

Data set so large takes long time to read and parse it. I decided to store it in PostgreSQL database. In the beginning both XML and binary sets were stored in one table, but then it was divided into two tables; then more support tables were added; PostgreSQL table inheritance was used to ease operating on main data tables[7].

Created database can be seen as temporal, and when looking at XML data also as spatial one. Such databases store information about presence of phenomenas in space and time. This database stores information about presence of tags (and probably persons wearing them) at the place at the moment. Also activities done to this tags, like pressing button, are stored. Additional spatial data, like geometry of building and rooms where events were held, and temporal data (schedule of Congress) can be used for more sophisticated analysis.

Created tables

station Describes readers

sputnik base table for storing data; tables with data inherit from it

ccc23 contains binary data from 23C3

ccc23xml contains XML data from 23C3; has additional columns containing values of attributes from XML file

reader table used to store data received by USB reader

adjacency stores count of readings seen by pairs of readers

room describes lecture rooms

event describes events that took place during 23C3; taken from Schedule XML file

[7]Scripts creating database can be downloaded from my web page

Base table for holding data from tags

id

time

sequence value of sequence counter

strength strength of signal

station id of station that received this signal

tags array of data, like pressed button

XML data table
is like raw data table and also contains:

position position of tag

plane position on the floor

direction direction; currently only (0. 0, 0)

observer

observedobject

priority

mindistance

maxdistance

Table of rooms
Describes room in which events (lectures) were taking places.

id identifier of room

name name of room: "Saal 1", "Shelter foo", ...

shape path describing room shape. Currently empty column; data to fill it could be taken from GPS data or from building plans

ymin

ymax

bbox Is it necessary, or better use geometry calculations or PostGIS?

Event table
Describes information about events. Populated using XML schedules published on http://www.ccc.de/

id identifier of event

organizerid

name name of event

place identifier of room event is taking place

description human-readable description

address URL of description of event

start timestamp of beginning moment of event

finish timestamp of end moment of event

Table containing data from 23C3 occupies about 700MB on hard drive. Data types used to store sequence and time values occupy 8 bytes each; index for each of those columns takes 250MB. Sequence identifier is stored as 4 byte integer and its index takes about 130MB. Creation of those indexes is necessary to have database offering good performance. This is not huge database, but is rather large for desktop computer.

Large amounts of rows can be changed when operations on data are performed. To be able to find good query plan, PostgreSQL needs to have accurate statistics of stored data. PostgreSQL does not update rows in place, but creates new row and marks old as deleted; this technique is called MultiVersion Concurrency Control (MVCC). So once in a while database needs to be vacuumed to remove all those deleted rows and to gather statistics. Autovacuum is daemon that takes care of observing all tables and performing vacuum when it is needed. Its default settings are too low for Sputnik data. The more reasonable is to analyse data table after 0.5% rows were changed and vacuum after 10% rows were changed. It makes sense to have more aggressive autovacuum by setting cost limit to 500 and delay to 0.

PostgreSQL client library, libpq, fetches entire result data set into RAM. This can be problem when exporting Sputnik data from database. I was getting “out of memory” error, so I had to use cursor to be able to retrieve data set partially. Solving this problem internally in libpq library (by using internal cursor) to be able to fetch large data set partially is in ToDo list of PostgreSQL.

5 Analysis of data

To understand further operations, one needs to understand how internally tags work. In each transmission tag sends its ID and strength of signal it uses to transmit. Each transmission is encrypted using XXTEA. To avoid replay attacks, it is necessary to change packets. Because adding real time clock would be too complicated, ever-increasing counter was added. Base station discards all packages with counter numbers less that the one seen previously. To avoid problems with reset of tag (removing battery) when counter is again set to 0, counter was divided. Higher word was saved on reset, and lower not. So after reset tag increases higher word, so counter value always grows. This feature means that gaps occur in counter values sequences when tag is reset. To avoid collisions, each tag transmits and sleeps random time, from 2 to 4 seconds.

5.1 Basic graphs

Following pictures present simple characteristics of data. They are based on work done by Peter Meerwald, mostly to make sure that data was correctly imported. Numbers present on following figures are larger than presented by Peter Meerwald. He was using hash tables to store Sputnik data, so he had not seen 64k repeated observations, which become visible in database.

Figure 1 presents how many packets were seen by more than one station. It shows only situations where stations were seeing more than 1000 common packets. It can be used to deduce how people were walking inside Congress Center, and also could be used to deduce positions of readers inside building.

Figure 2 shows number of packets seen in entire system in each minute. It can be seen that during day there is high activity, and during night hours activity is very low, because most of attended left the BCC.

Figure 3 shows activity of all XML data points. It shows both observations containing valid estimated position, and position “0, 0, 0”. Activity in the beginning consists of observations with invalid position; almost all later observations contain valid positions.

Following tables show number of packets that each reading station has received and number of received packets with particular strength of signal.

Packets read by each station

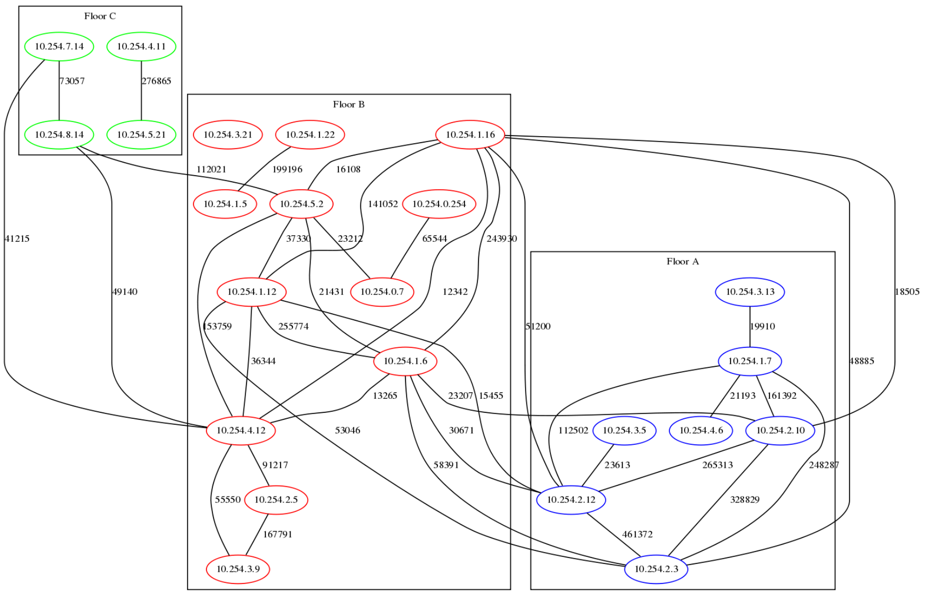

Figure 1: Pings read by more than one station (> 1000)

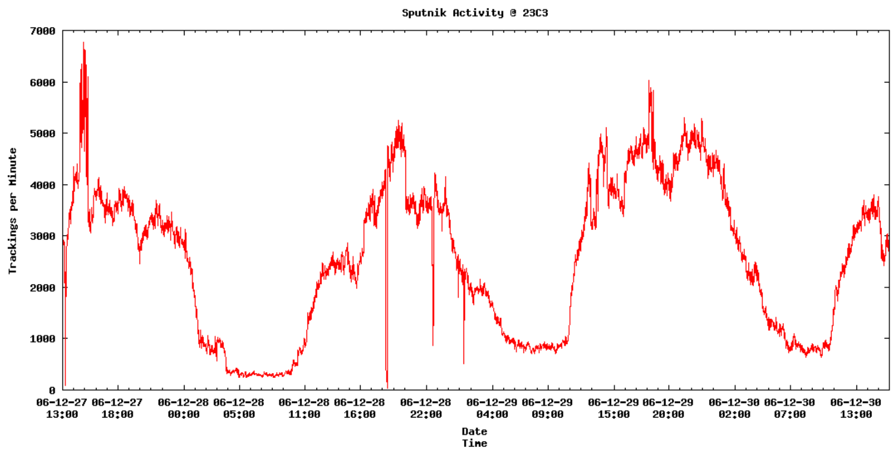

Figure 2: Number of packets read during one minute

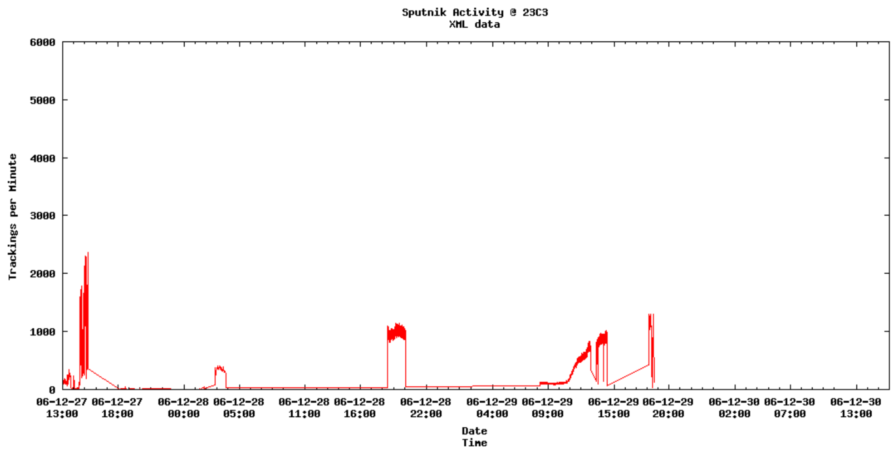

Figure 3: Number of packets read during one minute including unknown points

Id	IP address	count
2	10.254.2.3	1322696
21	10.254.5.21	880833
3	10.254.2.12	760606
15	10.254.1.6	758782
18	10.254.5.2	596466
14	10.254.4.12	589640
20	10.254.8.14	585443
26	10.254.1.16	570525
5	10.254.1.7	568765
4	10.254.2.10	563488
1	10.254.4.6	542657
16	10.254.1.12	532699
22	10.254.4.11	528187
11	10.254.1.22	494524
10	10.254.1.5	448760
9	10.254.2.5	428565
8	10.254.3.9	376396
24	10.254.3.5	231483
23	10.254.7.14	225075
17	10.254.0.254	187078
6	10.254.3.13	130379
13	10.254.0.7	129144
12	10.254.3.21	54863
25	10.254.0.100	8524

Strength of packets

Strength	count
0	182874
85	568413
170	1167287
255	9225658

5.2 Rebuilding sequences

To be able to analyse data and gain some knowledge from it, sequences need to be restored. It requires joining single packets into sequences and then attaching unique number into each found sequence. Unfortunately original tag identifiers are lost and it is impossible to recover them; but even without them restoring sequences will allow for analysis of data.

Global searching requires large amounts of CPU time, RAM and disk resources, so first program was using local search for short sequences.

Following snippet presents ideal situation when building sequences. It takes first packet and then tries to find next one, that has next value of counter, and is 1 or 2 seconds from previous one. It does not take into consideration gaps in sequences because of person leaving BCC, or because one is not in the range of any readers, or when tag is transmitting too weak signal to be received by any of readers. However it presents idea of finding local sequences; following functions are using this idea and add code dealing with gaps and choosing one packet that can be added to sequence when there is more than one.

First attempt of building sequences

```
SELECT time, extract('epoch' from time), sequence
FROM sputnik.sputnik WHERE id IS NULL AND
time BETWEEN %s::TIMESTAMP WITH TIME ZONE
AND %s::TIMESTAMP WITH TIME ZONE+%s::INTERVAL
for i in c.fetchall():
    old_e, old_s = int(i[1]), int(i[2])
    old_major = old_s/65536
    old_minor = old_s%65536
    p = []
    for j in data:
        e, s = int(j[1], int(j[1])
        major = s/65536
        minor = s%65536
        probable = (major == old_major and minor == old_minor+1)
            or (major == old_major+1 and minor == 0)
        if probable: p.append([e, s])
    if len(p) > 0:
        print old_e, old_s,
        for j in p: print j[0], j[1],
```

Basic idea of algorithm for searching local sequences is enhancements of code above. It takes all points from choosen period of few dozens seconds. To find all sequences of ticks there it assumes that ticks are about 1.5s from one another. Starting from the lowest counter value it tries to find the next value. In case of very close values of counter, difference of time is 1 or 2 seconds. In case of longer time distances, difference should be closer to 1.5s for every tick. It ignores data about strength of signal or stations that were able to receive it.

When more than one packet can be chosen to extend sequence conflict occurs, and this problem must be resolved. Conflict may be because either at the same time there are two different counter values, or the same value occurs at different moments. In case of either conflict we must choose only one packet to include in sequence, and discard another one. It needs to be noticed that not only two, but more packets may be involved in conflict. The general case is presence of more than one sub-sequence that can extend existing sequence. Only one of them must be chosen, as adding all sub-sequences will destroy existing sequence by introducing decreases in either time or counter values.

Sub-sequence may be chosen by taking into consideration length or resemblance to already existing sequence. Using separate function for choosing sequence to add allows for researching on different criteria of choosing and introducing more sophisticated criteria.

Alternative solution is creation of function returning next values of time and counter, basing on sequence that is being rebuilt. This is more complicated, as it requires knowing exact parameters of tag, especially time when it was started or reset, and exact time tag sleeps between transmissions.

Function `GetTickDistance` returns difference between counter values. It tries to take reset into consideration by treating reset as difference of 1. It decides that reset occurred when values passed as arguments have differing high words. However if there is less than about one minute to change of high word, it does not assume reset was involved.

Distance between sequence values

```
# Assumes a <= b
# Will not work when there is more than 1 overflow
def GetTickDistance(a, b):
    majora = a/65536
    minora = a%65536
    majorb = b/65536
    minorb = b%65536
# Inside one minor, or less than minute to overflow
    if majora >= majorb or minora >= 65500:
        return b-a
    else:
        return majorb-majora + minorb+1
```

To be able to recreate sequences it is necessary to create all alternatives and then choose the best ones. Hashes are used to store all counter values that were received at any moment, and all moments when any value of counter was received. All keys of hashes are read in increasing order, and all values stored under every key are considered as extensions of sequences. If considered point can be added to sequence, it is. If not, conflict is detected. Previous value is removed from sequence, and both points are added to special list of alternatives. In such case each subsequent point is treated as extension not of main sequence, but alternative sub-sequences. If it can be added to all of them, alternatives are stored, and this point is added to main sequence. If it can be added to only some of sub-sequences, conflict still remains. If it cannot be added to any of sub-sequences, it is added as another alternative sub-sequence.

Function `FindBestSequence` takes sequence and all alternative sub-sequences calculated by previous function and builds optimal sequence. It chooses the best possible sub-sequences to add. To choose the best ones it uses slope of sub-sequences, and chooses one with the slope closest to 1.5. Minimal square difference is used to find slope closest to ideal.

Finding best sequences amongst all created

```
# Sequence with len >= 3
def FindBestSequence(a):
    b = max(map(len, a))
    c, a = a, []
    for i in c:
        if len(i) == b: a.append(i)
# Find minimal difference between min and max, in case of many alternative sequences
    best = i = a[0]
    ds = float(i[1][0]-i[0][0])/GetTickDistance(i[0][1], i[1][1])
    mini = maxi = ds
    for j in range(1, len(i)-1):
        ds = float(i[j+1][0]-i[j][0])/GetTickDistance(i[j][1], i[j+1][1])
        mini = min(mini, ds)
        maxi = max(maxi, ds)
    c = (mini-1.5)*(mini-1.5)+(maxi-1.5)*(maxi-1.5)
    for i in a[1:]:
        ds = float(i[1][0]-i[0][0])/GetTickDistance(i[0][1], i[1][1])
        maxi = mini = ds
        for j in range(1, len(i)-1):
            ds = float(i[j+1][0]-i[j][0])/GetTickDistance(i[j][1], i[j+1][1])
            mini = min(mini, ds)
            maxi = max(maxi, ds)
        d = (mini-1.5)*(mini-1.5)+(maxi-1.5)*(maxi-1.5)
        if d < c: best, c = i, d
    return best
```

Described algorithm can be implemented in two ways. Main loop may iterate over time and check all possible counter values, or it can iterate over counter values and check all moments of appearance of this value. Those approaches should be equivalent, but iterating over counter values gives as result more and longer sequences. If using more CPU time is not a problem, both variants can be used and the best results given by any of them are chosen, independently for each considered interval.

First code that was used to use large part of data was implementation of $O(N^3)$ algorithm. For each point it was finding whether any of other points can be added to the sequence by checking if equation $\Delta s = a\Delta t, 1.0 \leq a \leq 2.0$ was met. After finding all possible points it was generating all possible alternatives from this chosen set. As it was checking all other points for every point from given interval, this operation was $O(N^2)$. If any sequence was found, it was removed from data set, and entire process was started from the beginning, thus $O(N^3)$ time cost.

$O(N^3)$ **algorithm**

```
SELECT DISTINCT time, extract('epoch' from time), sequence
FROM sputnik.sputnik WHERE id IS NULL AND
time BETWEEN %s::TIMESTAMP WITH TIME ZONE
AND %s::TIMESTAMP WITH TIME ZONE+%s::INTERVAL
a, b, again = 0, 0, True
while again:
    again, s = False, []
    for i in data:
        majort, majors = int(i[1]), int(i[2])
        p = [[majort, majors]]
        for j in data:
            minort, minors = int(j[1]), int(j[2])
            dt = minort-majort
            ds = GetTickDistance(majors, minors)
            if dt > 0 and ds <= dt and dt <= 2*ds:
                p.append([minort, minors])
        if len(p) > 1:
            again = True
            r = CreateAllSequencesSeqs(p)
            s = FindBestSequence(r)
            a += 1
            if len(s) > b: b = len(s)
            break
    if again:
        for i in s:
            UPDATE sputnik.sputnik SET id = %s
            WHERE sequence = %s AND time = to_timestamp(%s)
            for j in data:
                if i[0] == j[1] and i[1] == j[2]:
                    data.remove(j)
                    break
        id += 1
```

Improving speed of this algorithm came from observation that the longest sequences are be made when starting from the lowest time and lowest counter values. Query was changed to return sorted result. Algorithm was changed to take first tuple, and try to find all other tuples that can make sequence with the first one. If sequence was found, it was removed from data set; if not, only the first tuple was removed. So for each tuple all other tuples were considered, which gives $O(N^2)$. Because there is no repetition of this process if sequence is found, but further tuples are processed, this cost remains.

This algorithm gives the same results as previous one; this was proved by comparing sequences generated by both for few intervals. Cost of those algorithms can be slightly higher than $O(N^3)$ and $O(N^2)$ when considering building and comparing alternative sub-sequences. However size of such sub-sequences is small when compared to main sequences. Also size of sub-sequences tend to remain constant even when increasing length of analysed interval, which increases size of generated sequences.

$O(N^2)$ algorithm

```
SELECT DISTINCT time, extract('epoch' from time), sequence
FROM sputnik.sputnik WHERE id IS NULL AND
time BETWEEN %s::TIMESTAMP WITH TIME ZONE
AND %s::TIMESTAMP WITH TIME ZONE+%s::INTERVAL
ORDER BY sequence, time
a, b = 0, 0
while len(data) > 0:
    s, i = [], data[0]
    majort, majors = int(i[1]), int(i[2])
    p = [[majort, majors]]
    for j in data[1:]:
        minort, minors = int(j[1]), int(j[2])
        dt = minort-majort
        ds = GetTickDistance(majors, minors)
        if dt >= 0 and ds <= dt and dt <= 2*ds:
            p.append([minort, minors])
    if len(p) > 1:
        r = CreateAllSequencesSeqs(p)
        s = FindBestSequence(r)
        a += 1
        if len(s) > b: b = len(s)
        for j in s:
            UPDATE sputnik.sputnik SET id = %s
            WHERE sequence = %s AND time = to_timestamp(%s)
            for k in data:
                if j[0] == k[1] and j[1] == k[2]:
                    data.remove(k)
                    break
            id += 1
        else:
            data.remove(i)
```

Function JoinIDs computes all sequences for one interval and interval after that, and then tries to join found sequences. For each sequence in main interval it calculates coefficient of line created by its last point and by first point of sequence from the next interval. If any line with coefficient between 1.0 and 2.0 is found it means that those sequences are candidates for joining. However they would also have to have the same coefficients themselves before they could be joined.

Function trying to join found sequences

```
def JoinIDs(c, t, d, period):
    main = GetLines(c, t.strftime("%Y-%m-%d %H:%M:%S+01:00"), period)
    after = GetLines(c, (t+d).strftime("%Y-%m-%d %H:%M:%S+01:00"), period)
    before = GetLines(c, (t-d).strftime("%Y-%m-%d %H:%M:%S+01:00"), period)
    for i in sorted(main.keys()):
        majort = main[i]['max-time']
        majors = main[i]['max-seq']
        for j in sorted(after.keys()):
            minort = after[j]['min-time']
            minors = after[j]['min-seq']
            dt = minort-majort
            ds = GetTickDistance(majors, minors)
            if ds <= dt and dt <= 2*ds:
                print "Can Join"
                print "\t", main[i]['id'], main[i]['length'], main[i]['min-time'], main[i]['min-seq'],
                print main[i]['max-time'], main[i]['max-seq']
                print "with", ds, dt, float(dt)/ds
                print "\t", after[j]['id'], after[j]['length'], after[j]['min-time'], after[j]['min-seq'],
                print after[j]['max-time'], after[j]['max-seq']
```

I think it could be even possible to improve local algorithm to have $O(N)$ time cost. However it was not implemented so I do not know if it is really possible and if it would give good results.

Function calculating distance in counter values was changed, as it was producing strange sequences (65600, 132000, 512000, . . .). Reset was ignored, and distance was ordinary difference of counter values. However this was not helpful. Local algorithms were not able to find long enough sequences. Although few found sequences were rather long (up to 20 packets for 1 minute), but most found were only consisting of 2 or 3 packets. This was leading to large gaps between sequences from consecutive intervals, and troubles with joining them.

New distance in sequence counter function

```
# Assumes a <= b
# Will not work when there is more than 1 overflow
def GetTickDistance(a, b):
    majora = a/65536
    minora = a%65536
    majorb = b/65536
    minorb = b%65536
    return b-a
```

Scatter plots drawn for long intervals are revealing straight lines. This lead to the idea to find straight lines (as drawn in geometry) and to treat them as sequences. To avoid problems with reset calculations were done inside 64k blocks of counter values.

The best way to find the longest sequences is to start with point with the lowest values of counter and time. Then try to draw lines through it and all other points from the range. Choosing slope that results in line going through the most points gives the longest sequence. This is greedy algorithm as in each step the largest sequence is chosen.

To choose the best line coefficient histogram of all slopes is used, with bucket of size 0.1. To be sure that no point is left because of rounding errors, range of slopes is used: all points that are on lines with slopes differing less than ± 0.3 from chosen slope are included into created sequence.

Because for each point all other points are used to calculate slopes and then all points that are in right coefficient range are chosen, time cost is $O(N^2)$.

It finds long sequences. It leaves only about 4000 points (out of 11.1 million) without any sequence. However rather strange line coefficients are found; besides ordinary 2.4, 2.5, it comes with 0.1, 0.4, 0.5, 9.9, 10.0, 8.1, …

Function `FindIDs` takes range of counter values and tries to find all sequences in this range. It finds all counter values and for each value finds all times it occurs; this is similar to hashes used in local algorithms. Then for each starting point histogram of all coefficients of lines is created and the largest value is used. Query similar to one calculating slopes is used to mark all points as belonging to one sequence. Update is done by one SQL query.

Finding sequences in global manner

```
def FindIDs(connection, sa, sz, ta, tz, id):
    SELECT DISTINCT sequence FROM sputnik.sputnik WHERE id IS NULL
    AND sequence BETWEEN %s AND %s ORDER BY sequence
    for s in c.fetchall():
        s0 = s[0]
        SELECT DISTINCT time FROM sputnik
        WHERE id IS NULL AND sequence = %s
        for t in
            t0, hash = t[0], {}
            SELECT DISTINCT ON (sequence, time) time, sequence,
            (extract('epoch' FROM (time-%s)))::float/(sequence-%s)::float
            FROM sputnik.sputnik WHERE id IS NULL AND time > %s AND
            sequence BETWEEN %s AND %s AND sequence != %s
            ORDER BY sequence, time
            for i in  c.fetchall():
                k = int(i[2]*10)
                if 0 < k and k <= 100:
                    hash[k] = hash.get(k, 0)+1
                i = c.fetchone()
            k = -1.0
            if len(hash) > 0:
                m = max(hash.values())
                for i in sorted(hash.keys()):
                    if m == hash[i]:
                        k = float(i)/10.0
                        break
                UPDATE sputnik.sputnik SET id = %s WHERE id IS NULL
                AND sequence = %s AND time = %s
                UPDATE sputnik.sputnik SET id = %s WHERE id IS NULL AND
                sequence BETWEEN %s AND %s AND sequence != %s AND
                (extract('epoch' FROM (time-%s)))::float/(sequence-%s)::float
                BETWEEN %s AND %s
                id += 1
    return id
```

Following code shows calling of function for creating sequences. First the lowest unused value for identified sequence is found, and then function `FindIDs` is called for each of the values of high word of tag counter. First range was divided into time intervals so program operates on smaller data sets, but because of error in code time interval was not respected and first call calculated all sequences from entire range.

Calling a sequence finder

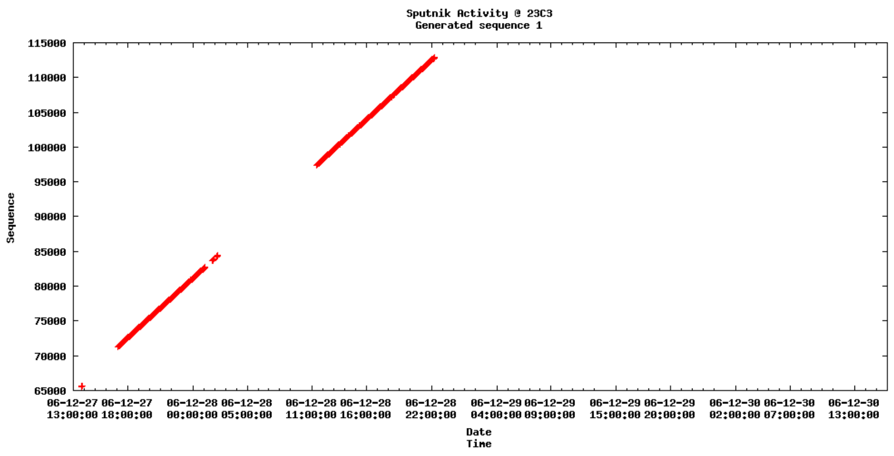

Figure 4: Generated sequence; first set, number 1

```
id = (SELECT MAX(id) FROM sputnik.sputnik WHERE id IS NOT NULL)+1
ta = '2006-12-27 12:59:19+01:00'
tz = '2006-12-30 15:59:59+01:00'
id = FindIDs(connection, 0, 2*65536, ta, tz, id)
# Very large data set, 2924448 rows
id = FindIDs(connection, 131072, 196608, '2006-12-27 12:59:19+01:00', '2006-12-27 18:00:00+01:00', id)
id = FindIDs(connection, 131072, 196608, '2006-12-27 18:00:00+01:00', '2006-12-28 00:00:00+01:00', id)
id = FindIDs(connection, 131072, 196608, '2006-12-28 00:00:00+01:00', '2006-12-28 17:00:00+01:00', id)
id = FindIDs(connection, 131072, 196608, '2006-12-28 17:00:00+01:00', '2006-12-29 00:00:00+01:00', id)
id = FindIDs(connection, 131072, 196608, '2006-12-29 00:00:00+01:00', '2006-12-29 16:00:00+01:00', id)
id = FindIDs(connection, 131072, 196608, '2006-12-29 16:00:00+01:00', '2006-12-30 00:00:00+01:00', id)
id = FindIDs(connection, 131072, 196608, '2006-12-30 00:00:00+01:00', '2006-12-30 15:59:59+01:00', id)
# Very large data set, 2076875 rows
id = FindIDs(connection, 3*65535, 4*65536, ta, tz, id)
# Very large data set, 1277488 rows
id = FindIDs(connection, 4*65535, 5*65536, ta, tz, id)
# Very large data set, 1016195 rows
id = FindIDs(connection, 5*65535, 6*65536, ta, tz, id)
# Very large data set, 620763 rows
id = FindIDs(connection, 6*65535, 7*65536, ta, tz, id)
```

Figures 4 to 9 show sequences generated by this algorithm. Some sequences are the proper ones, but other are wrong; their points really belong to many different sequences.

Figures 5 and 6 show sequences that from the beginning look like collage of many sequences. They show the main problem of algorithm: range of allowed coefficients is too wide, and too many points are added to sequence. The farther away from the first point, the more obvious it is.

Figure 7 shows sequence that in the beginning is correct, and gets wrong only in the end. So first part should be preserved, and after it, somewhere is this gap, sequence should end.

Figure 8 shows sequence that is generated by all variants of global algorithm.

Sequence shown in Figure 9 shows errors that came from integer overflow. Because initially I did not use Python large integers, counter values close to 4 billions were treated as small negative values, and joined with real small values. Column storing counter values was using 64-bit integers, so PostgreSQL was able to update rows with large counter values, and not destroy other sequences.

Figure 10 shows packets that were not used in any sequence. It was only about 4000 points, and it's very good result for data set consisting of 11.1 million of points.

Figure 11 shows size of generated sequences calculated as number of occurrences of pair (time, counter value); event if packet was seen by more than one reader, it was counted only once. In other words it shows number of occurrences of tag, not how many times it was seen.

Figure 12 shows size of sequences calculated as number of tuples that are included into each sequence.

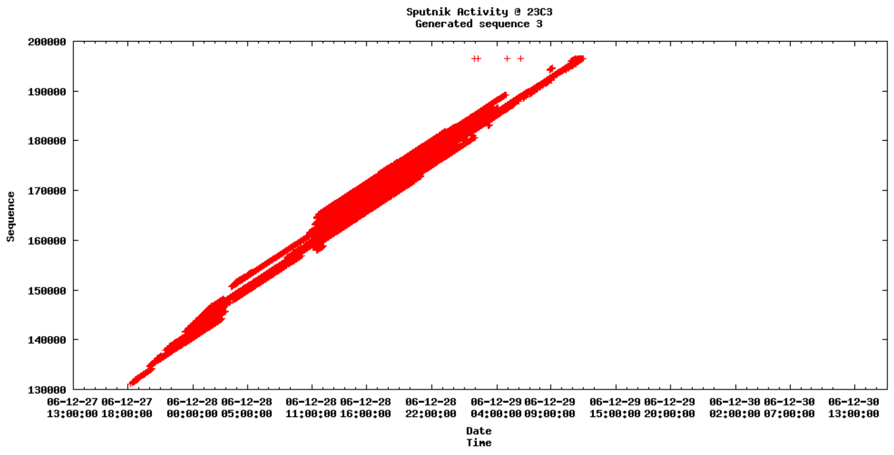

Figure 5: Generated sequence; first set, number 3

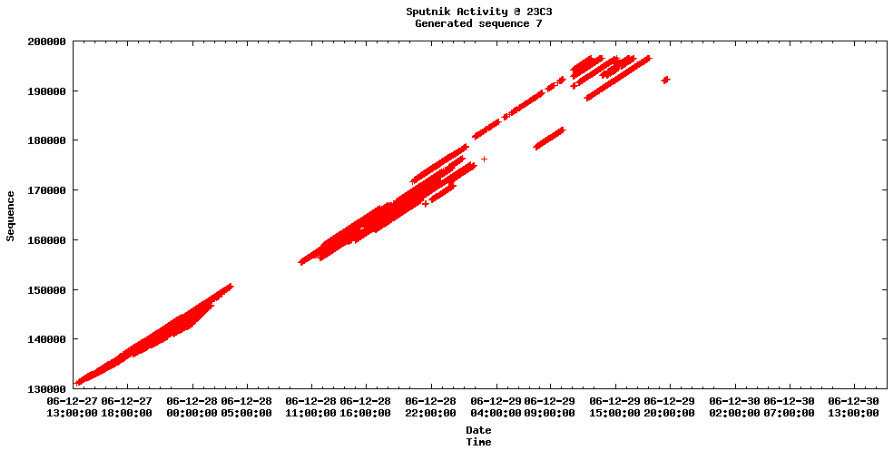

Figure 6: Generated sequence; first set, number 7

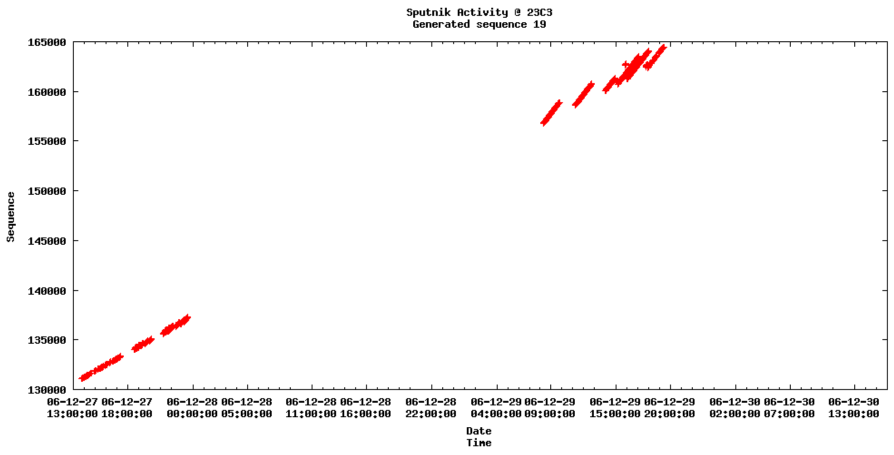

Figure 7: Generated sequence; first set, number 19

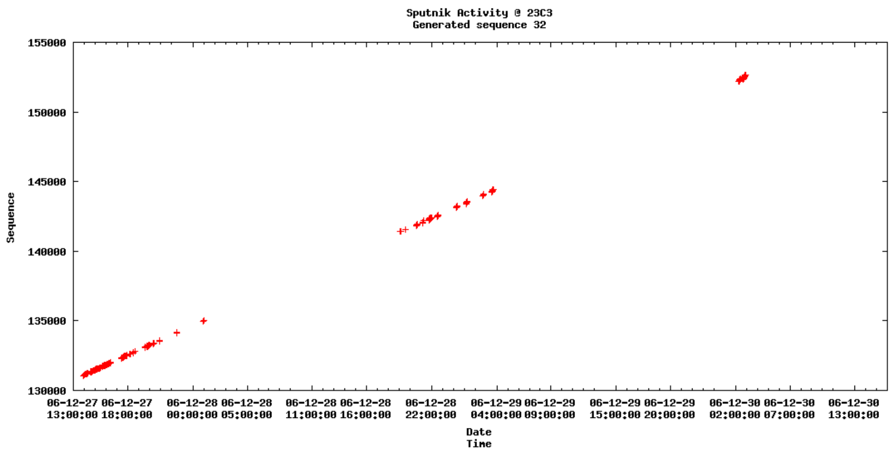

Figure 8: Generated sequence; first set, number 32

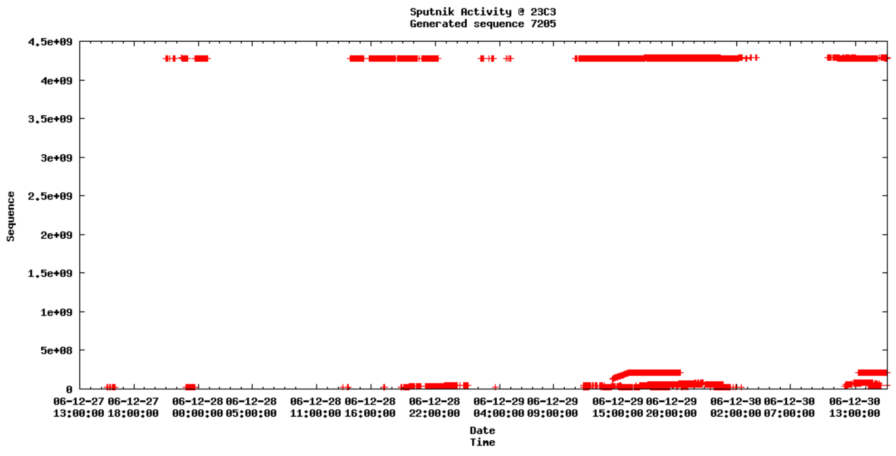

Figure 9: Generated sequence; first set, number 7205

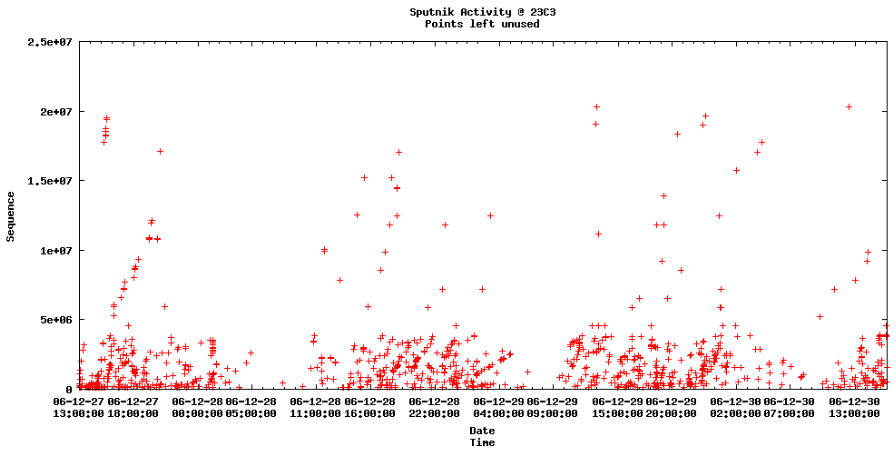

Figure 10: Points left without sequence; first set

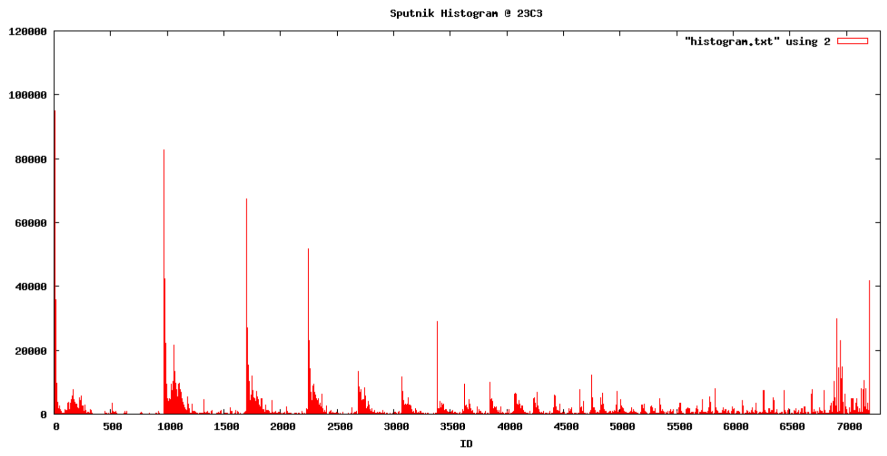

Figure 11: Histogram of sizes of generated sequences for the first set

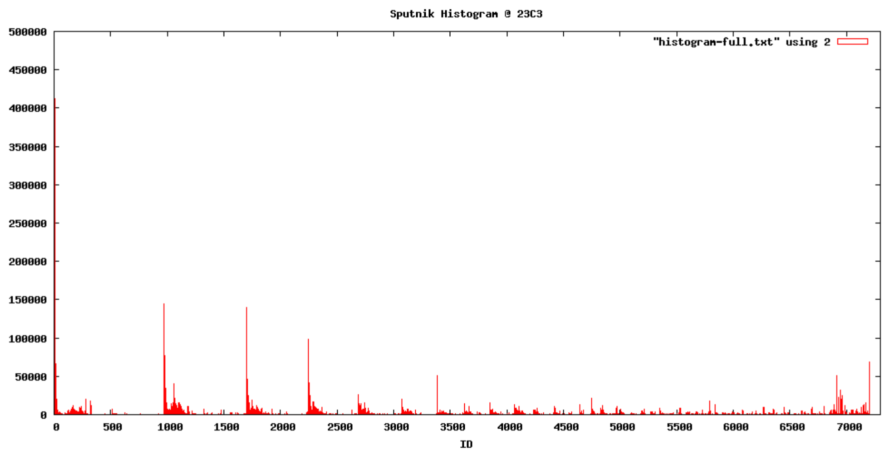

Figure 12: Histogram of sizes of generated sequences for the first set

Program was running for about 72h on AMD Duron 1.3GHz with 768MB RAM and single HDD IDE 7200RPM. It was IO-constraint, probably because of database size larger than available RAM; CPU was not much used. Clustering data table according to counter values could improve performance in the beginning. However PostgreSQL does not try to preserve clustering, so after adding many points to sequences clustering would be lost and Input/Output capacity would again become limiting factor. Also PostgreSQL decides to scan entire table if there is more than 5% rows in result, so in this algorithm entire data table is read.

The main problem with algorithm are sequences that contain point that should belong to many different sequences. This is caused by too wide range of possible coefficient values. The more distant from the initial point, the more visible the problem is.

Figure 13 shows histogram of line coefficients for buckets of size of 0.1. Figure 14 shows histogram of line coefficients for buckets of size of 0.001. As can be seen, first histogram presents false situation; number of points in many lines that consist of small number of points but have close coefficient values is able to outnumber one line with high number of points. So in this situation instead of long one line short one is chosen, and all its neighbours that were able to outnumber the long ones are joined to this improper sequence.

Improvements of algorithm were necessary to get better results. First was refactoring of code; most of activities were moved into functions. Second improvement was creation of SQL aggregate function to choose only one counter value at any given time. This function was used together with grouping with respect to time, and chosen point was the closest one to the chosen slope. To avoid problems with many

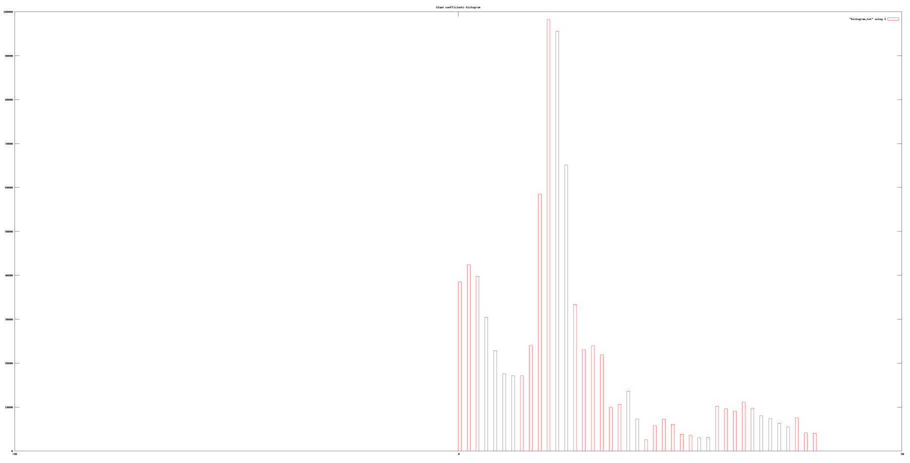
Figure 13: Coefficients histogram for 10 buckets

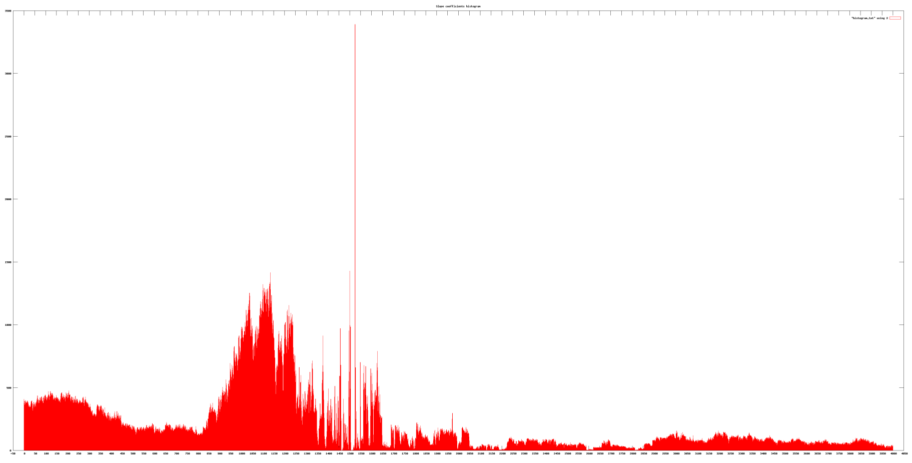
Figure 14: Coefficients histogram for 1000 buckets

lines joining into one width of histogram buckets was changed to 0.001. Histogram was calculated for slopes from range 1.0 to 5.0. Additionally range of allowed coefficients was changed from ±0.3 to ±0.001. However this caused gap at the beginning of each sequence; because of rounding errors in the first few minutes slope was not close enough to the ideal to be included in chosen range of slopes.

Function `sputnik_guessbest` is SQL aggregate used to choose one point in case of presence of more than one counter value at the same time. It requires grouping by time in SQL query. It chooses point which distance from the chosen slope is the smallest. To be able to calculate distance from this line it needs to know parameters of line; before using this aggregate function `sputnik_guessinit` must be called. Initialisation function must be called before every query using `sputnik_guessbest`. Both functions are written in pl/Python and use global hash for PostgreSQL Python functions to store line parameters and the best found point.

Currently PostgreSQL in Debian does not offer trusted pl/Python, so untrusted pl/PythonU is used. Creation of functions in untrusted languages requires administrative access to database (usually user "postgres") and SECURITY DEFINER during creation to allow ordinary used to use it.

Grouping function

```
CREATE OR REPLACE FUNCTION sputnik.guessinit(t TIMESTAMP WITH TIME ZONE, sequence BIGINT, slope DOUBLE PRECISIO
RETURNS VOID
VOLATILE RETURNS NULL ON NULL INPUT SECURITY DEFINER
LANGUAGE 'plpythonu' AS
```

```
$$
GD["time"] = t
GD["sequence"] = sequence
GD["slope"] = slope
$$;

CREATE OR REPLACE FUNCTION sputnik.guessbest(state BIGINT, t TIMESTAMP WITH TIME ZONE, sequence BIGINT)
RETURNS BIGINT
VOLATILE CALLED ON NULL INPUT SECURITY DEFINER
LANGUAGE 'plpythonu' AS
$$
if (not GD.has_key("time")) or (not GD.has_key("sequence")) or (not GD.has_key("slope")):
return None
if (t is None) or (sequence is None):
return None

plan = plpy.prepare("""
SELECT (extract('epoch' FROM ($1::TIMESTAMP WITH TIME ZONE-$2::TIMESTAMP WITH TIME ZONE)))::float/($3::BIGINT-$
""", ["timestamptz", "timestamptz", "int8", "int8"])

result = sequence
if state is not None:
r0 = plpy.execute(plan, [t, GD["time"], sequence, GD["sequence"]], 1)
r1 = plpy.execute(plan, [t, GD["time"], state, GD["sequence"]], 1)
if abs(r0[0]["slope"]-GD["slope"]) >= abs(r1[0]["slope"]-GD["slope"]):
result = sequence
else:
result = state
return result
$$;

CREATE AGGREGATE sputnik.guesser (TIMESTAMP WITH TIME ZONE, BIGINT) (
SFUNC = sputnik.guessbest,
STYPE = BIGINT
);
```

Function Histogram calculates histogram of slopes of all lines going through given point. If there is more than one slope with the same maximal number of points, the smallest one is chosen. Function returns slope and number of points in bucket. If it is unable to calculate any slope it returns pair 0, 0.

Histogram function

```
def Histogram(c, time, sequence, sa, sz):
    hash = {}
    c.execute("""SELECT DISTINCT ON (time, sequence) time, sequence,
    (extract('epoch' FROM (time-%s::TIMESTAMP WITH TIME ZONE)))::float/(sequence-%s::BIGINT)::float
    FROM sputnik.sputnik WHERE id IS NULL AND
    sequence BETWEEN %s::BIGINT AND %s::BIGINT AND
    time > %s::TIMESTAMP WITH TIME ZONE AND
    sequence > %s::BIGINT""", (time, sequence, sa, sz, time, sequence))
    i = c.fetchone()
    while i != None:
        k = int(i[2]*1000)
        if 1000 <= k and k <= 5000:
            hash[k] = hash.get(k, 0)+1
        i = c.fetchone()
    if len(hash) > 0:
        m = max(hash.values())
        for i in xrange(1000, 5001):
# Let's take the smallest max
            if m == hash.get(i, 0):
                result = float(i)/1000.0
                break
        return result, m
    else:
        return 0.0, 0
```

Function `Line` takes as parameters starting point of line, slope of line and allowed range of slopes and finds all points that lie on that line. It initialises global Python hash, as main query uses aggregate `sputnik_guessbest`. It retrieves all matching points from database and returns list holding them.

Function finding points on line with given slope

```
def Line(c, time, sequence, slope, margin, sa, sz):
    result = [[time, sequence]]
    c.execute("""SELECT sputnik.guessinit(%s::TIMESTAMP WITH TIME ZONE,
    %s::BIGINT, %s::DOUBLE PRECISION)""", (time, sequence, slope))
    c.execute("""SELECT time, sputnik.guesser(time, sequence)
    FROM sputnik.sputnik WHERE id IS NULL AND
    sequence BETWEEN %s::BIGINT AND %s::BIGINT AND
    time > %s::TIMESTAMP WITH TIME ZONE AND
    sequence > %s::BIGINT AND
    (extract('epoch' FROM (time-%s::TIMESTAMP WITH TIME ZONE)))::float/(sequence-%s::BIGINT)::float
    BETWEEN %s::float AND %s::float GROUP BY time
    ORDER BY time""", (sa, sz, time, sequence, time, sequence, slope-margin, slope+margin))
    i = c.fetchone()
    while i != None:
        result.append([i[0], i[1]])
        i = c.fetchone()
    return result
```

Function `FindIDs` iterates through all values of counter inside given range, and finds all times when any counter had particular value. Each such pair is treated as potential starting point of line; histogram of slopes is calculated, and if returned bucked holds more than 8 points, new sequence is created. Unlike previous version, this function does not use one update query, but every point is updated by separate SQL command.

Function finding all lines

```
def FindIDs(connection, sa, sz, id):
    c.execute("""SELECT DISTINCT sequence
    FROM sputnik.sputnik WHERE id IS NULL AND
    sequence BETWEEN %s AND %s
    ORDER BY sequence""", (sa, sz))
    start = c.fetchall()
    for s in start:
        s0 = s[0]
        c.execute("""SELECT DISTINCT time FROM sputnik.sputnik
        WHERE id IS NULL AND sequence = %s""", (s0,))
        for t in c.fetchall():
            t0 = t[0]
            slope, count = Histogram(c, t0, s0, sa, sz)
            if slope > 0.0 and count >= 8:
                line = Line(c, t0, s0, slope, 000.1, sa, sz)
                for i in line:
                    UPDATE sputnik.sputnik SET id = %s WHERE id IS NULL AND
                    time = %s::TIMESTAMP WITH TIME ZONE AND
                    sequence = %s::BIGINT
                id += 1
    return id
```

Figures 15 to 19 show sample sequences generated by improved algorithm.

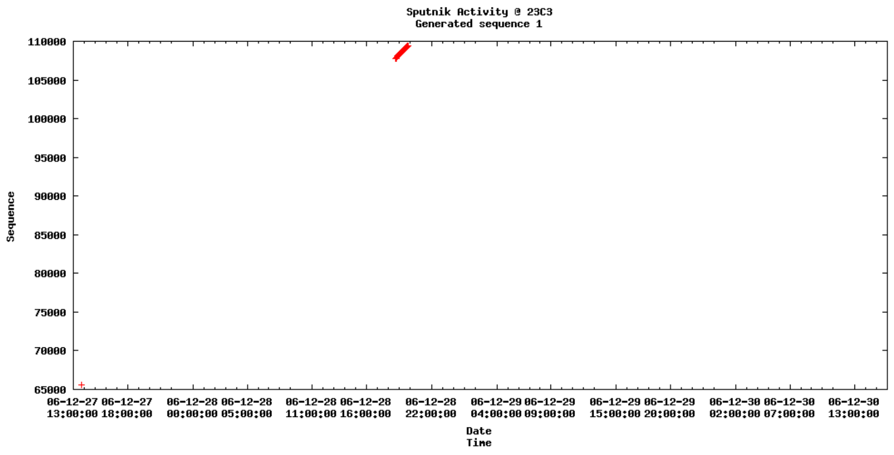

Figure 15: Generated sequence; second set, number 1

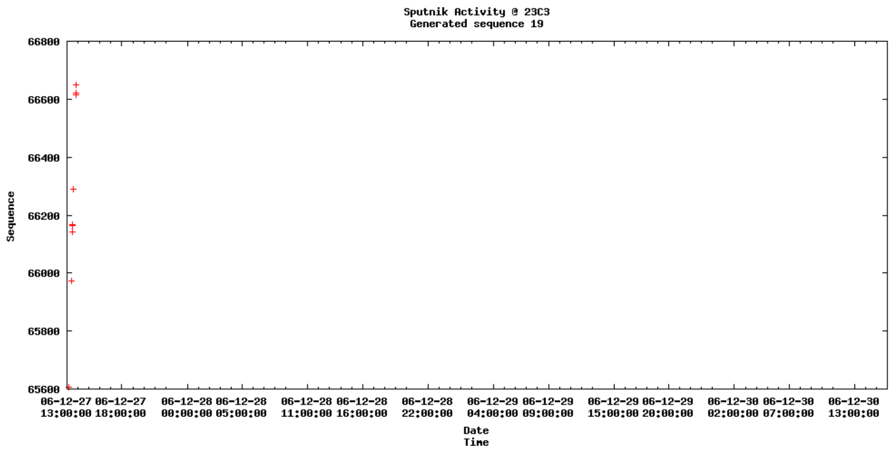

Figure 16: Generated sequence; second set, number 19

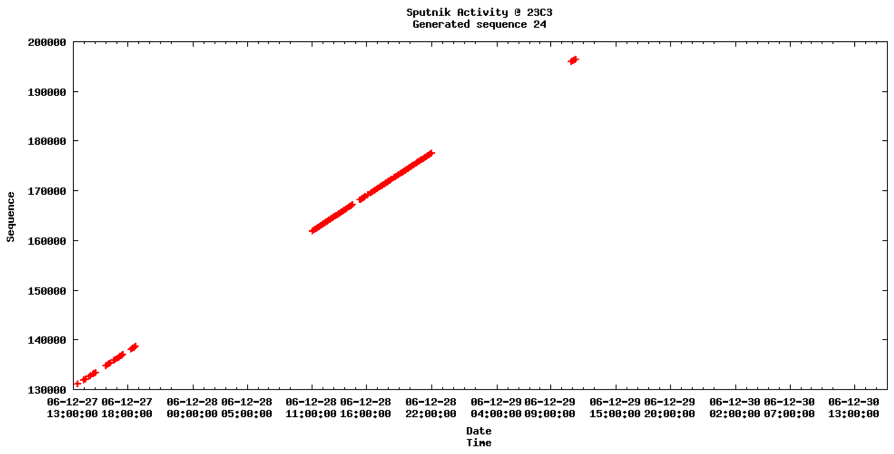

Figure 17: Generated sequence; second set, number 24

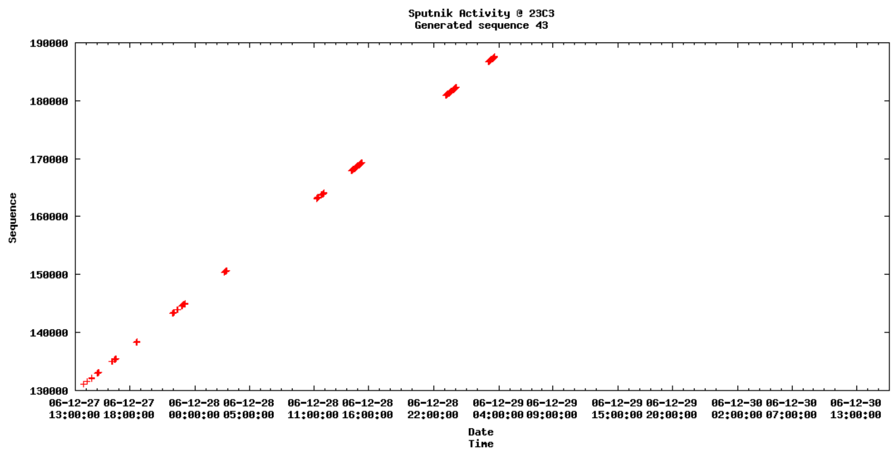

Figure 18: Generated sequence; second set, number 43

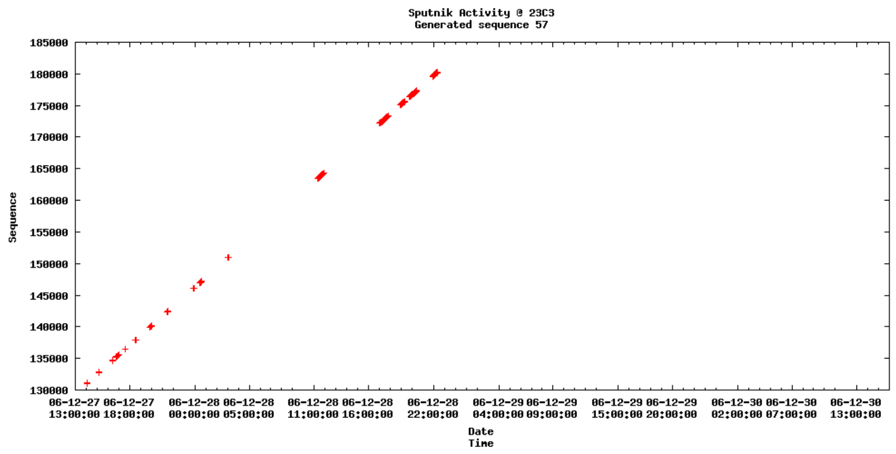

Figure 19: Generated sequence; second set, number 57

Figure 17 shows sequence that is generated by all variants of global algorithm.

Figures 18 and 19 shows generated sequences that have missing some points. Either program did not add some points that should be taken into those sequences or persons wearing those tags was appearing and disappearing from sight of readers.

Figure 20 shows size of generated sequences calculated as number of occurrences of pair (time, counter value); event if packet was seen by more than one reader, it was counted only once. In other words it shows number of occurrences of tag, not how many times it was seen.

Figure 21 shows size of sequences calculated as number of tuples that are included into each sequence.

Program was running very slowly. It was running for almost 2 weeks before I interrupted it. It could not go outside first large data set ($counter \in < 2*65536; 3*65536 >$) so I stopped program and run it for later counter values. It did not leave the next counter values block. It was using IO subsystem and CPU more equally. Its slow speed may come from performing more calculations, using pl/Python function, and updating information about sequences as many individual queries instead of one bulk query.

Generated sequences were initially big, but later they were getting smaller and smaller, down to dozen points.

Algorithm was joining sequences in spite of aggregate function which was used to guard against it. Data analysis was showing that some sequences had errors, but as they were more subtle it was not easily seen on the graphs,

Figure 22 shows two distinct sequences that are joined. Their points are in allowed slope range, and their packets are interlaced, so even aggregate function can not remove one of them.

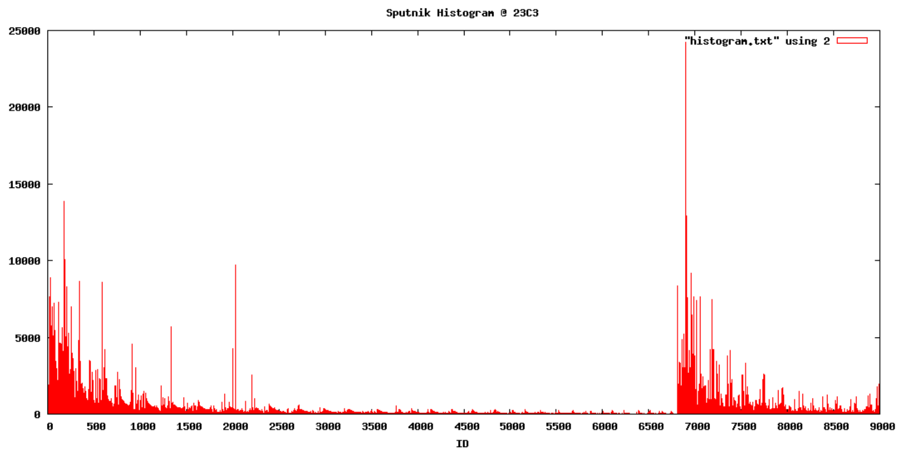

Figure 20: Histogram of sizes of generated sequences for the second set

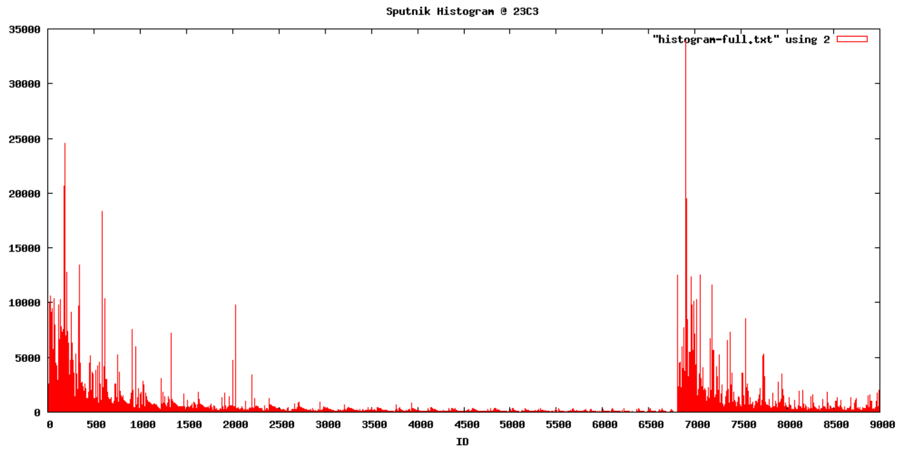

Figure 21: Histogram of sizes of generated sequences for the second set

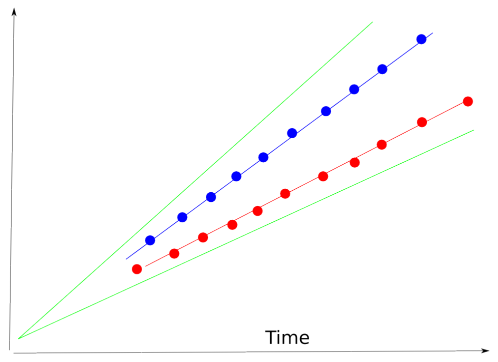

Figure 22: Interlaced sequences

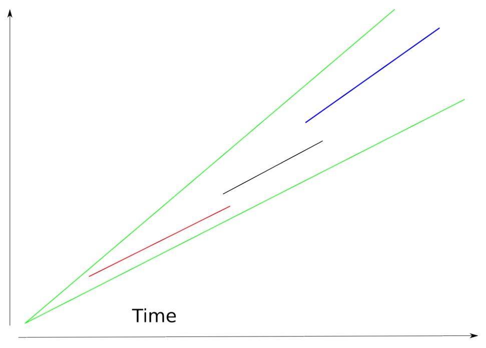

Figure 23: Collinear sequences

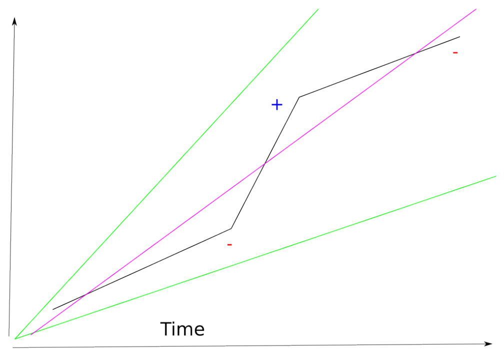

Figure 24: Incorrectly joined sequences

Figure 23 shows three distinct sequences joined into one. They have similar slope and their points lie in allowed range, so they are joined together, even though that points should create distinct sequences.

Figure 24 shows three sequences that have different slopes, but are also joined. This situation can be detected by calculating difference of slopes between consecutive points, similarly to differentiating. The long sequence of differences of the same sign may mean followed by long sequence of differences of another sign suggests join of different sequences.

Figure 25 shows sequence that have points not placed directly on ideal line. It may seem similar to previous situation, but (especially if differences between points and slopes are not large) it is single sequence. The main difference between situation in figures 24 and 25 is number of points that have the same sign of difference between slopes and absolute difference between those slopes. If both of those parameters are small, there is single sequence.

New firmware of tags was released during CCC2007. Transmission was not occurring every few seconds, but about 10 times a second. This, together with USB reader, allowed for analysing if discarding sub-second parts introduces large error in scope of lines. I took few minutes of readings, and calculated two slopes, one taking all data into consideration, and another using floor function to discard milliseconds. Resulted slopes differed on 4th place after comma, so having only seconds when transmission occurred does not result in error disallowing operating on data.

Either having too wide range and having joined sequences, or having too narrow range and leaving some points out, without guarantee that appropriate points are included in sequence meant need for including additional data in searching for good sequences. First of additional variables that could point whether to include tuple into the sequence was signal strength. Each tag changes strength of sent signal, either in sequence of 0x00, 0xff, 0x55, 0xff, 0xaa, 0xff, 0xff, 0xff, or in 0x00, 0x55, 0xaa, 0xff, depending on used firmware version.

First problem would be that in old firmware 5 out of 8 values was 0xff, so it would be difficult to determine where in sequence of signal strengths particular point is. However analysing of source code

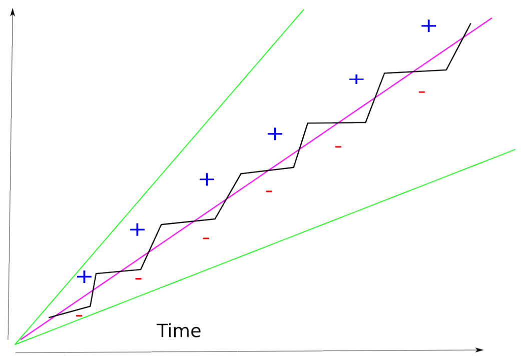

Figure 25: Correctly joined sequence

and Sputnik data revealed that strength of signal was not distinctive between tags. Each tag starts at the same strength sequence point, so there is no variability between sequences. If more than one point has the same counter value, they also have the same strength of signal. It can not be used to distinguish different sequences.

As mentioned earlier, because of rounding errors at the beginning of sequence coefficients do not have the same values as coefficients for further points. It is necessary to have wider allowed range of slopes in the beginning and more narrow near the end. This can be accomplished by sigmoid function[8]. Function $0.01 + \frac{0.09}{1+e^{(x-500)/100}}$ was used in program. At the distance 0 it generated border of 0.1; its value was getting smaller to reach 0.01 for argument of 1000. Because of very large exponential values, FPU exception was generated for arguments greater than about 70000.

Because strength of signal could not be used, stations that received signal from tag were used. The main assumption was that set of seen stations did not change from one point to another if that points were close in time. To keep algorithm simple only list of seen stations was considered, not their distribution in space. Similarity was defined as number of stations in both sets, divided by size of joined sets.

If strengths of signals in both points differ similarity function was slightly changed, and returned number of stations seen using weaker signal divided by number of stations seen with stronger signal. But because most of points in data set had the strongest value of signal, there was not many situations with different signals between points.

To avoid errors shown in Figures 22, 23, and 24, algorithm was changed to retrieve all potential points that could be added to generated sequence and choose the best one itself. This approach is return to the idea of generating alternative sub-sequences used in local algorithm.

Points that are in conflict have condition $\neg(T_1 > T_0 \land S_1 > S_0)$ met. Program creates all possible sub-sequence from them and then chooses the best one. To choose the best it locally compares lengths, slopes of sub-sequences and reading stations seen by all sub-sequences and chooses one that is the most similar to main sequence.

Last version of algorithm differs from previous ones, and those changes can be summarised in "take more points and choose the best ones". Instead of using constant range, sigmoid function was used to include more points in the beginning of sequence. All points are read from database, and program builds alternative sequences from them. Instead of using custom aggregate function to choose only one point, standard function aggregating all seen stations into array is used. This array is then used to choose the best points to include into sequence. The last change is breaking line if it is discovered that created line has high probability of being two different lines.

Function `Similarity` returns number from range $< 0.0; 1.0 >$. This is degree of similarity of two sets of readers that were able to receive signal from tag. Function uses sets introduced in Python 2.4.

Similarity of seen stations

```
def Similarity(a, b):
    result = 0.0
    station0, strength0 = a
```

[8] http://en.wikipedia.org/wiki/Sigmoid_function

```
    station1, strength1 = b
    size0, size1 = len(station0), len(station1)
    if strength0[0] > strength1[0]:
        same = 0.0
        for i in station1:
            if i in station0: same += 1
        result = same/len(station1)
    elif strength0[0] < strength1[0]:
        same = 0.0
        for i in station0:
            if i in station1: same += 1
        result = same/len(station0)
    else:
        result = float(len(set(station0)&set(station1)))/
            float(len(set(station0)|set(station1)))
    return result
```

Function Fetch reads all points from database that can be used to create sequence. It takes all packets that were received less than two minutes after first point of sequence, and then returns those which slope lies in range determined by sigmoid function.

Getting all points that can create line

```
def Fetch(c, time, sequence, slope, sa, sz):
    result = [[time, sequence, slope, 0.0]]
    c.execute("""SELECT sputnik.array_accum(station),
    sputnik.array_accum(strength)
    FROM sputnik.ccc23 WHERE id IS NULL AND
    time = %s::TIMESTAMP WITH TIME ZONE AND
    sequence = %s::BIGINT""", (time, sequence))
        i = c.fetchone()
    if i != None:
        result[0].append(i[0])
        result[0].append(i[1])
    i = c.fetchall()
# Union of first 100s and the rest
    c.execute("""SELECT time, sequence,
    (extract('epoch' FROM (time-%s::TIMESTAMP WITH TIME ZONE)))::float/(sequence-%s::BIGINT)::float,
    0.0, sputnik.array_accum(station), sputnik.array_accum(strength)
    FROM sputnik.ccc23 WHERE id IS NULL AND
    sequence > %s::BIGINT AND sequence <= %s::BIGINT+100::BIGINT AND
    time > %s::TIMESTAMP WITH TIME ZONE AND time <= %s::TIMESTAMP WITH TIME ZONE+'100 second'::INTERVAL
    GROUP BY time, sequence
    UNION
    SELECT time, sequence,
    (extract('epoch' FROM (time-%s::TIMESTAMP WITH TIME ZONE)))::float/(sequence-%s::BIGINT)::float,
    0.0, sputnik.array_accum(station), sputnik.array_accum(strength)
    FROM sputnik.ccc23 WHERE id IS NULL AND
    sequence BETWEEN %s::BIGINT AND %s::BIGINT AND
    time > %s::TIMESTAMP WITH TIME ZONE AND
    sequence > %s::BIGINT AND
    (extract('epoch' FROM (time-%s::TIMESTAMP WITH TIME ZONE)))::float/(sequence-%s::BIGINT)::float
    BETWEEN %s::float-sputnik.BorderWidth(sequence-%s) AND %s::float+sputnik.BorderWidth(sequence-%s)
    GROUP BY time, sequence ORDER BY time""", (time, sequence, sequence, sequence, time, time, time, sequence,
    i = c.fetchone()
    while i != None:
        result.append([i[0], i[1], i[2], i[2]-result[-1][2], i[4], i[5]])
        i = c.fetchone()
    return result
```

Function `Lines` takes list of all points that were read from database and creates all possible sequences from them. It is similar to function used in local algorithm.

Calculating all possible sequences from points

```
def Lines(data):
    result = [] candidate = []
    for i in data:
        num = 0
        for j in candidate:
            if i[0] > j[-1][0] and i[1] > j[-1][1]:
                num += 1
        if len(candidate) == num:
            if len(candidate) == 1: result.extend(candidate[0])
            elif len(candidate) > 1: result.append(candidate)
            candidate = [[i]]
        else:
            for j in candidate:
                if i[0] > j[-1][0] and i[1] > j[-1][1]:
                    j.append(i)
            if 0 == num: candidate.append([i])
# Add last alternative
    if len(candidate) == 1: result.extend(candidate[0])
    elif len(candidate) > 1: result.append(candidate)
    return result
```

Function `Line` takes all sub-sequences and chooses the best line from all given alternatives. Each of alternatives has calculated up to five factors that are taken into consideration: length, similarity of slopes in the beginning and in the end, similarity of seen stations in the beginning and in the end. Only the best sub-sequence gets points for each factor, and then only the best one is chosen. If there is more than one best alternative, the first one is chosen.

The very important part of this function if condition $j[0][0] > result[-1][0]\ldots$ which allows only sub-sequences which time and counter values are greater than already existing in sequence to be considered as alternatives. This protects from the problem of having improper sequence in case when one alternative choosing after another.

Choosing the best line from all alternatives

```
def Line(lines):
    result = []
    for i in xrange(len(lines)):
        if type(lines[i][0]) != type([]): result.append(lines[i])
        else: alternatives = []
            if len(result) > 0:
                for j in lines[i]:
                    if j[0][0] > result[-1][0] and j[0][1] > result[-1][1]: alternatives.append(j)
            else: alternatives = lines[i]
            scores = [0] * len(alternatives)
            sizes = map(lambda x: len(x), alternatives)
            best = max(sizes)
            for j in xrange(len(alternatives)):
                if sizes[j] == best: scores[j] += 1
            stationsa = map(lambda x: Similarity((result[-1][4], result[-1][5]), (x[0][4], x[0][5])), alternati
# Find best alternative for stations in the beginning
            if i+1 < len(lines) and type(lines[i+1][0]) != type([]):
                stationsz = map(lambda x: Similarity((x[-1][4], x[-1][5]), (lines[i+1][4], lines[i+1][5])), alt
# Find best alternative for stations in the end
            slopesa = map(lambda x: abs(alternatives[x][0][3]-result[-1][3]), xrange(len(alternatives)))
# Find best alternative for slopes in the beginning
            if i+1 < len(lines) and type(lines[i+1][0]) != type([]):
                slopesz = map(lambda x: abs(alternatives[x][0][3]-lines[i+1][3]), xrange(len(alternatives)))
```

```
# Find best alternative for slopes in the end
# Find the best alternative:
            best = max(scores)
            for j in xrange(len(alternatives)):
                if scores[j] == best:
                    result.extend(alternatives[j])
                    break
# Count slope deltas once more, for final line proposal
    slope = result[0][2]
    for i in result:
        i[3] = i[2]-slope
        slope = i[2]
    return result
```

Function `Break` takes four consecutive points a, b, c, and d and returns number from range $< 0.0; 1.0 >$, the probability that line should be broken between points b and c, because they belong to different lines. It takes six factors into consideration: difference in slopes between lines a-b and b-c, and b-c and c-d, difference in time between following points, similarity of seen stations between points b and c, and absolute changes of slope between local and global value.

Function returning probability of break

```
def Break(a, b, c, d, slope):
    result = 0.0
    SlopeDiff = 10.0
    SlopeTrigger = 0.01
    CounterDiff = 100
    TimeDiff = datetime.timedelta(0, 120)
    StationSimilarity = 0.5
    if abs(c[3]) > SlopeTrigger:
        if abs(c[3]) > abs(b[3])*SlopeDiff: result += 1.0
        if abs(c[3]) > abs(d[3])*SlopeDiff: result += 1.0
# Time is more intuitive that sequence counter
# Also I do not have to think about line coefficient
#    if c[1] - b[1] > CounterDiff: result += 1.0
    if c[0] - b[0] > TimeDiff: result += 1.0
    if Similarity((b[4], b[5]), (c[4], c[5])) < StationSimilarity: result += 1.0
    SlopeAB = float((b[0]-a[0]).seconds)/(b[1]-a[1])
    SlopeBC = float((c[0]-b[0]).seconds)/(c[1]-b[1])
    SlopeCD = float((d[0]-c[0]).seconds)/(d[1]-c[1])
# Slopes should be similar to each other and to the main slope
    if slope-1.0 <= SlopeAB and SlopeAB <= slope+1.0 and (SlopeBC < slope-1.0 or slope+1.0 < SlopeBC):
        result += 1.0
    if slope-1.0 <= SlopeCD and SlopeCD <= slope+1.0 and (SlopeBC < slope-1.0 or slope+1.0 < SlopeBC):
        result += 1.0
    return result/6.0
```

Main function `FindIDs` calls all previous functions and generates sequence. It decides to break line if probability returned by function `Break` is more than 0.5, in such case of iteration of loop creates more than one sequence.

Function creating all lines

```
def FindIDs(connection, sa, sz, id):
    c.execute("""SELECT DISTINCT sequence FROM sputnik.ccc23 WHERE id IS NULL AND
    sequence BETWEEN %s AND %s ORDER BY sequence""", (sa, sz))
    for s in c.fetchall():
        s0 = s[0]
        c.execute("""SELECT DISTINCT time FROM sputnik.ccc23
        WHERE id IS NULL AND sequence = %s""", (s0,))
        for t in c.fetchall():
```

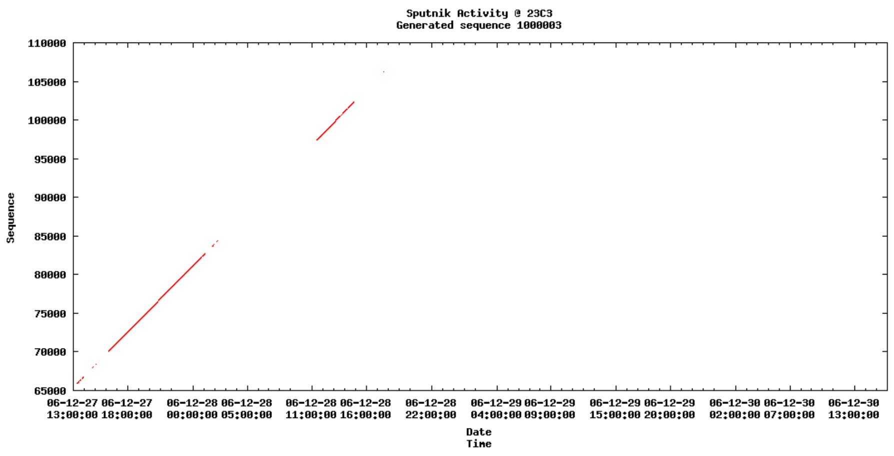

Figure 26: Generated sequence; third set, number 3

```
        t0 = t[0]
        slope, count = Histogram(c, t0, s0, sa, sz)
        if slope > 0.0 and count >= 8:
            data = Fetch(c, t0, s0, slope, sa, sz)
            lines = Lines(data)
            line = Line(lines)
            for i in xrange(len(line)):
                skip = False
                if len(line[i][4]) != len(line[i][5]):
                    print "Error in size of ", line[i]
                    skip = True
                s = line[i][5][0]
                for j in line[i][5]:
                    if j != s:
                        print "Error in strength of ", line[i]
                        skip = True
                if skip:
                    break
                UPDATE sputnik.sputnik SET id = %s WHERE id IS NULL AND
                time = %s::TIMESTAMP WITH TIME ZONE AND sequence = %s::BIGINT
                if i > 0 and i < len(line)-2:
                    b = Break(line[i-1], line[i], line[i+1], line[i+2], slope)
                    if b > 0.5:
                        id += 1
                        print "Break here, new id ", id, b
            id += 1
return id
```

Figures 26 to 30 show some of sequences generated by improved algorithm.

Figure 27 shows sequence that is generated by all variants of global algorithm.

Figure 31 shows size of generated sequences calculated as number of occurrences of pair (time, counter value); event if packet was seen by more than one reader, it was counted only once. In other words it shows number of occurrences of tag, not how many times it was seen.

Figure 32 shows size of sequences calculated as number of tuples that are included into each sequence.

Program was run on different machine than previous ones. It was running 5634 minutes on 64 bit AMD 3400+ with 1GB of RAM and one IDE HDD 7200RPM. It was stopped by FPU error in sigmoid function for large values of counter. 10.6 million rows was used in generated sequences. Over 1600 sequences were made from more than 1000 points.

Because many of generated sequences were short, the next step should be joining of them. One solution is to try to join existing sequences, another could be trying to extend sequences by points not

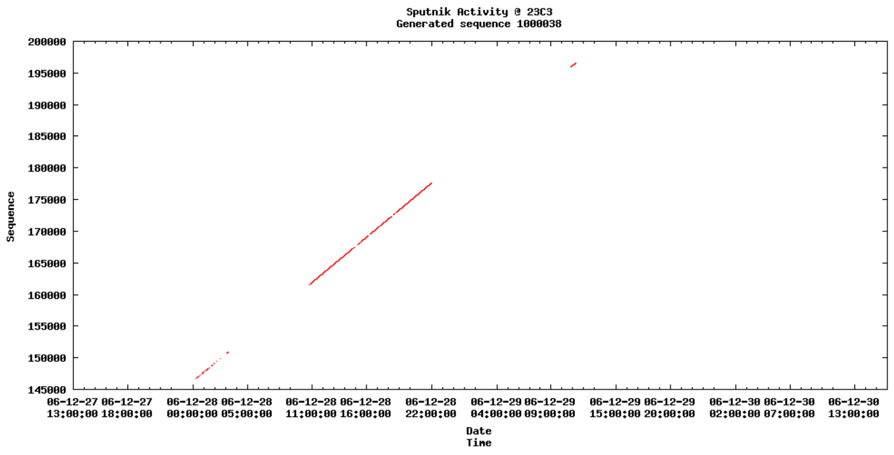

Figure 27: Generated sequence; third set, number 38

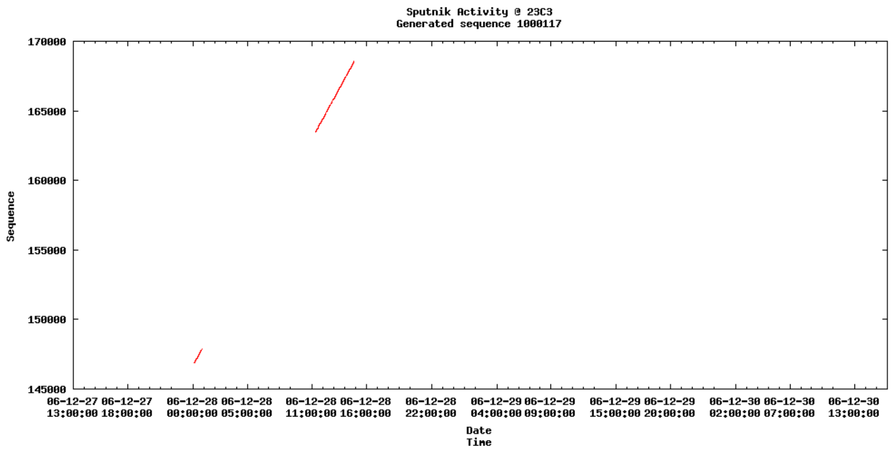

Figure 28: Generated sequence; third set, number 117

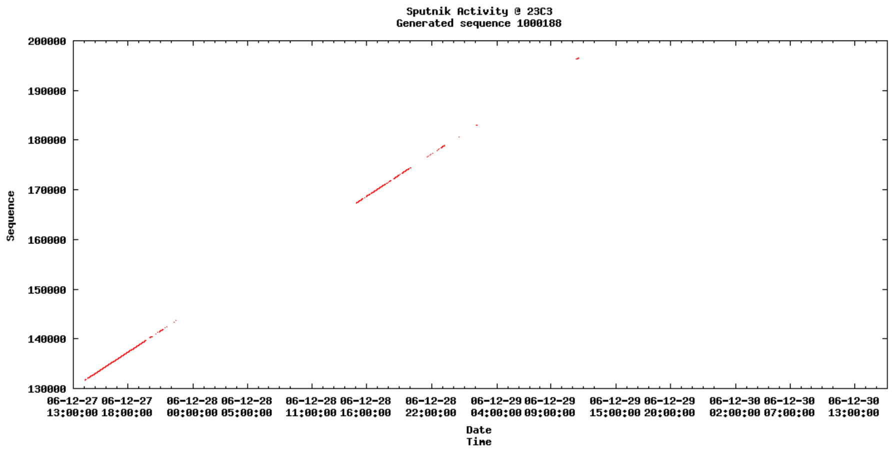

Figure 29: Generated sequence; third set, number 188

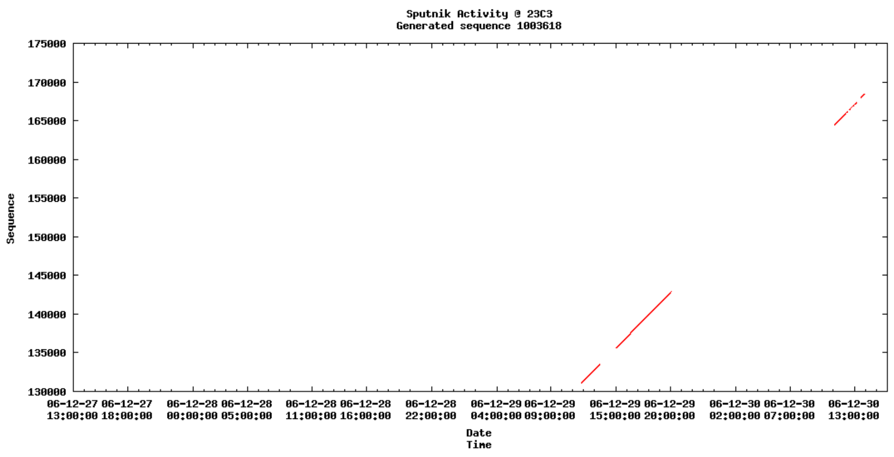

Figure 30: Generated sequence; third set, number 3618

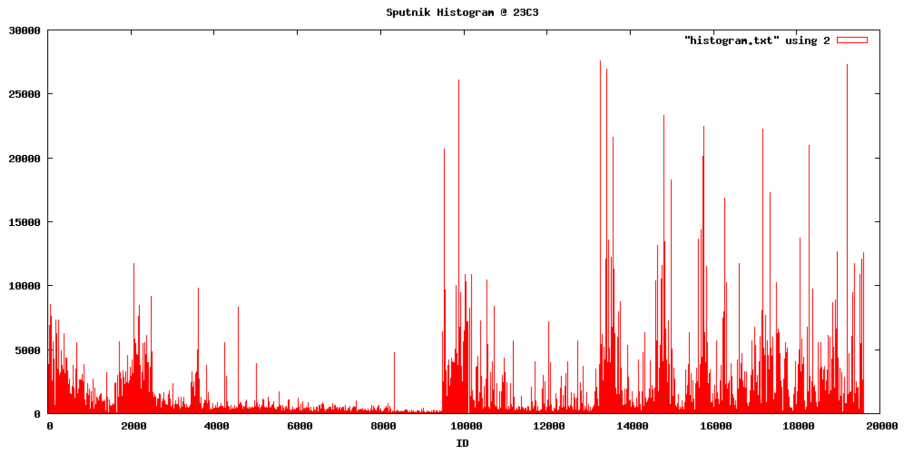

Figure 31: Histogram of sizes of generated sequences for the third set

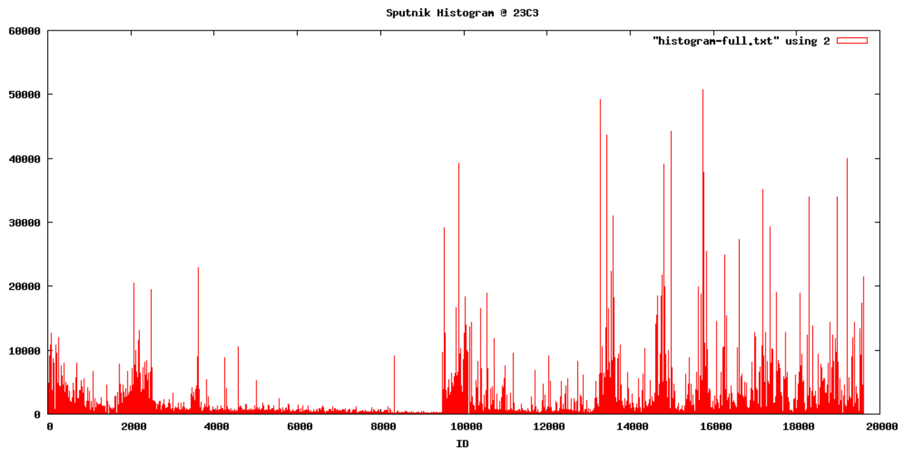

Figure 32: Histogram of sizes of generated sequences for the third set

belonging to any sequence. But problem with joining is choosing which sequence to join with each another. Which sequence from those shown in Figures 26, 27, 28. 29 should be joined to the one shown in Figure 30? It could be different case of `Break` function. If none of the causes for break occurs, there is possibility of join. Another possible solution is manual joining. Program could display few candidates and let user choose which ones look best together. If manual joining is success, this approach could be used to change generating algorithm and allow for manual choosing of alternative sub-sequences.

Knowledge gathered during analysing data and generating sequences leaves some doubts. I started with assumption that each tag sends packet every 1.5s. This lead to setting coefficient range from 1s to 2s. Because this was not giving good results in local algorithms, and by observing scatter plots, global algorithms were using range from 0.0 to 10.0, and later, basing on analysing source code of Sputnik firmware, from 1.0 to 5.0, Source code of firmware contains two calls of sleep function. One sleeps for 2s, and another for random period from 0s to 2s. This gives range of line slopes from 2s to 4s. But because second sleep function parameter is random value, there should be no straight line! However scatter plots reveal many of them. So either Sputnik data contains so many points that one can draw any line, or function rand() returns not very random numbers. Basing on analysing packets generated by single tag, second possibility is true.

Fragment of firmware of tag

```
void main (void)
{// get random seed
  ((unsigned char *) &seq)[0] = EEPROM_READ (4);
  ((unsigned char *) &seq)[1] = EEPROM_READ (5);
  ((unsigned char *) &seq)[2] = EEPROM_READ (6);
  ((unsigned char *) &seq)[3] = EEPROM_READ (7);
// increment code block after power cycle
  ((unsigned char *) &crc)[0] = EEPROM_READ (8);
  ((unsigned char *) &crc)[1] = EEPROM_READ (9);
  store_codeblock (++crc);
  seq ^= crc;
  srand (crc16 ((unsigned char *) &seq, sizeof (seq)));
// increment code blocks to make sure that seq is higher or equal after battery change
  seq = ((u_int32_t) crc) << 16;
  i = 0;
  while (1) {
// update code_block so on next power up the seq will be higher or equal
      crc = seq >> 16;
      if (crc == 0xFFFF) break;
      if (crc == code_block) store_codeblock (++crc);
// encrypt my data
      shuffle_tx_byteorder ();
      xxtea_encode ();
      shuffle_tx_byteorder ();
// send it away
      nRFCMD_Macro ((unsigned char *) &g_MacroBeacon);
      CONFIG_PIN_LED = 1; nRFCMD_Execute (); CONFIG_PIN_LED = 0;
// reset touch sensor pin
      TRISA = CONFIG_CPU_TRISA & ~0x02; CONFIG_PIN_SENSOR = 0;
sleep_jiffies (0xFFFF);
      CONFIG_PIN_SENSOR = 1; TRISA = CONFIG_CPU_TRISA;
// sleep a random time to avoid on-air collosions
sleep_jiffies (rand ());
      i++;
    }
}
```

No physical (or geometrical) model was taken into consideration during generating sequences. No distance between stations or speed of movement was analysed. This could give better results in sequences, by limiting point to only those that are in range to reach from previous point. On the other hand this approach would require calculating position of each tag in every moment.

5.3 Analysis

Following paragraphs describe potential approaches. They base on validity of generated sequences. I did not yet performed any analysis of data using generated sequences, as recovering them was my primary concern.

XML data set proves that it is possible to calculate position of tag. Tags send packets with different signal strength to allow for estimation of distance from reader. This estimation bases on negative knowledge. If reader is unable to read signal with small strength it means that tag is far away from it. So having few packets it is possible to calculate minimal and maximal distance tag is from reader. Power of signal was set so next level of power increases twice radius of range. This gives two spheres with small and large radius; person is between them. When data from few readers is known, it is possible to calculate common fragment of space where all those spheres intersect, and this is position of tag. But this requires knowing exact positions of readers.

Human body decreases strength of signal. This decreases precision of estimating position of tag. But maybe this could be used to calculate direction person has, assuming that tag is worn in the front. Range would not be sphere, but two hemispheres, larger in the front and smaller in the back. This would require performing more calculations (two times for each reader), but as there is no situation when all readers see one tag, it would not be impossible. Direction could be proven when person moves in this direction, again with assumption that person walks forwards, not backwards.

Simple analysis is calculating time of entering BCC and leaving it. Most people leave Center for the night, but some stay. Also when one sequence disappears and another one appears in the same place it means that someone is playing with battery and reset tag.

The most interesting analysis is looking for connections and similarities between attendees. This can be done by looking for people that attended similar talks. Those people may not even know each other but have common interests.

Another research area is looking for friends. Friends can be defined as people that stay together; they tend to be together not only during talks, but also and especially during breaks. If two people are close during most breaks, they are close friends. If they are close for some times, and not close for other moments, they may be colleagues. Or they may just stay in the same queue for pizza. However here the most important is relative position (distance between people), not exact position of tags.

This data set leves many conclusions to be drawn.