Helsinki University of Technology
Laboratory of Telecommunications Technology

January 13, 1999

Traffic in modem pools of Helsinki University of Technology

Jani Lakkakorpi
jlakkako@cc.hut.fi

Preface

The capacity of HUT staff modem pool was 120 simultaneous connections (2 * Portmaster 3) until the end of April 1998. Since then the staff pool capacity has been 60 connections (only one Portmaster 3).

The student modem pool capacity was 60 simultaneous connections (48 modem connections and 12 ISDN -connections) until the end of April. Since then the pool capacity has been 180 connections (3 * Portmaster 3).

The figures from October 1997 and January 1998 are from the old system and the figures from May, June and September 1998 are from the new system. In addition to these changes in pool capacities, the pricing policy of Helsinki Telephone Corporation has changed (18.3.1998): now the evening (17.00-07.00) and weekend rates are no longer fixed (47 p./call) - you have to pay 5.5 p. extra for every minute that exceeds 30 minutes…

Last summer I did some research about the traffic in these two pools [1]. The log files were from October 1997 - before the changes. It would be very interesting to see what kind of an effect these changes have on traffic in these modem pools. Do people now hang up after 30 minutes and call again? Has the student pool now enough capacity?

Contents

1. Log files

The modem pool log files are text files (created in the HUT computing center by Kimmo Laaksonen) with the following structure (it has changed a bit since I analyzed the log files of October 1997 last summer [1]):

- Each line contains the information of a single connection
- Comma is the field separator
- Information fields starting from left are:

1. Starting time of a connection YYMMDDHHMMSS
2. Weekday (0=sunday, 1=monday etc.)
3. Encrypted user ID
4. Session duration (sec.)
5. Bytes in
6. Bytes out
7. Caller phone number (3 last digits removed) or 99999 if unknown
8. Port type code:
   - Async = 0 (=modem)
   - ISDN = 2
   - ISDN-V120 = 3
   - ISDN-V110 = 4
9. Termination cause code:
   - User request = 1
   - Lost carrier = 2
   - Lost service = 3
   - Idle time-out = 4
   - Session time-out = 5
   - Admin reset = 6
   - Admin reboot = 7
   - Port error = 8
   - NAS error = 9
   - NAS request = 10
   - NAS reboot = 11
   - Port unneeded = 12
   - Port preempted = 13
   - Port suspended = 14
   - Service unavailable = 15
   - Callback = 16
   - User error = 17
   - Host request = 18
10. Initial connection speed etc. (if known) e.g. 33600 LAPM/V42BIS

Usually, a log file contains the user information of one month. This seems to be an appropriate amount of information to be processed at one time. Here is a small piece from the beginning of the student modem pool log file (September 1998):

```
980825093930,2,xwISNWQ6sxM,617818,0,0,99999,0,0,
980825093939,2,xwISNWQ6sxM,718083,0,0,99999,0,0,
980827142659,4,xwISNWQ6sxM,528041,0,0,99999,0,0,
980901062155,2,7t9wdkSzUBY,3404,975412,3067655,99999,0,1,
980901062812,2,LNKacPlzS3c,5281,167988,541033,095485,2,1,64000
980901063622,2,xRn3rvSSsqo,1374,256662,2634077,99999,0,1,
```

2. AWK-scripts

AWK-scripts are needed to extract the desired information from the log files so that Matlab-functions will be able to process that information. (For example starting times, holding times and weekdays).

AWK-script is formed of three parts. The first part is BEGIN-part, where some initializations are carried out. (For example date routines.)

The second part (BODY), is carried out for every line of the source file. For example: in *extract_times.awk* we change the starting time of a connection to seconds using 00:00:00 January 1st 1970 as offset time. Starting and termination times are stored in an array called *events*. Starting time is marked with +1 and termination time (starting time + holding time) is marked with -1. Now we have a list of arrivals and departures. Because AWK-arrays are not "real" arrays, we don't have to worry about the huge dead time since 1970.

In the last part (END) we go trough the data gathered and find the time for the first and the last event. Then we count the new offset time for this data so that it is the start of that day (00:00:00), where the first event lies. Finally the script prints out all the events (+1 or -1) with new times (seconds from the start of the first day).

Scripts can be used in the following way:

```
[jlakkako@keskus jansep98]$ awk -f extract_times.awk < 9809 > sepstudents.times
```

Before this file can be used successfully in Matlab we have to sort the file and then remove the first three lines (Min, Max, Newoffset etc.):

```
[jlakkako@keskus jansep98]$ sort -n < sepstudents.times > sepstudents.sortedtimes
```

This file now includes the moments of the events and the information whether the event was an arrival or a departure. It looks like this:

```
1880    1
21178   1
23371   1
```

AWK-scripts *extract_times.awk* ja *extract_data.awk* are modified versions of the scripts written by Esa Hyytiä [2].

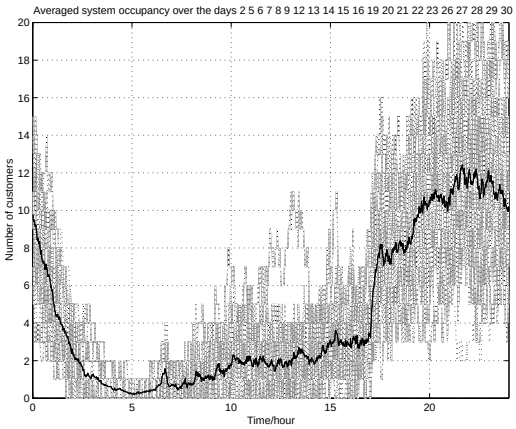
3. Processing data with Matlab

3.1. System occupancy averages

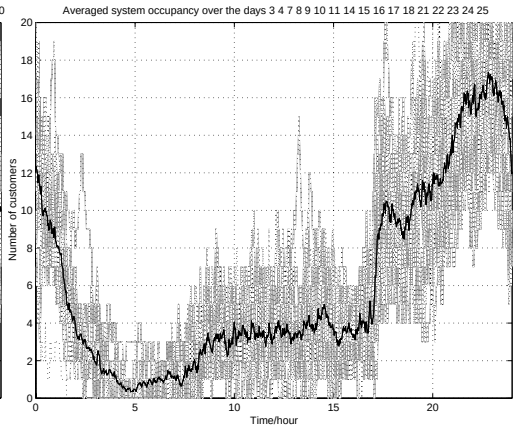
System occupancy is the number of customers at the pool in a particular moment of time. Matlab-function *events.m* counts and prints the system occupancy during the measurement period. It takes two arguments: the moments of time and the information whether the event was an arrival or a departure. Because the behavior of the system can be considered periodical, we can compute and print out the system occupancy average over selected days. The traffic in these days should be quite similar - for example we can compute the system occupancy average for weekdays and weekends (Sat. & Sun.). We use a Matlab-function *average.m* to compute and print the system occupancy average over the selected days. Functions *events.m* and *average.m* are written by Esa Hyytiä [2].

By using the following macro *sep_stu_ave_w.m* we can easily (just by typing sep_stu_ave_w in Matlab) compute and print out the weekdays' system occupancy average in September in the student modem pool. To see the variation within different days, all days are first plotted with broken lines, and finally the average line is plotted with a continuous line.

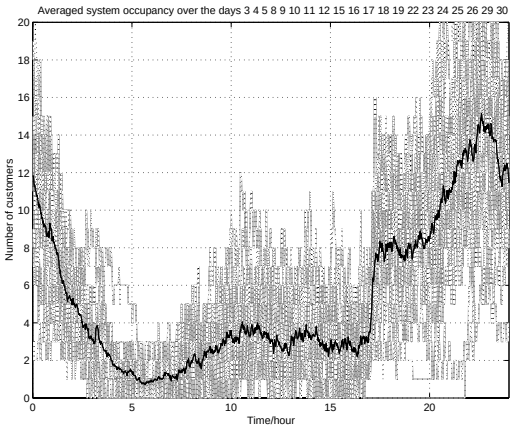
```
load sepstudents.times;
[ x, y ] = events( sepstudents(:,1), sepstudents(:,2) );
for i=10:36,
  if ( i ~= [ 12 13 19 20 26 27 33 34 ] )
    [ xi, yi ]  = average( x, y, [i] );
    plot( xi, yi, 'w' );
    plot( xi, yi, 'c:' );
    hold on;
  end;
end;
[ x0, y0 ] = average( x, y, [ 10 11 14 15 16 17 18 21 22 23 24 25 28
29 30 31 32 35 36 ] );
axis([0, 24, 0, 120]);
plot( x0, y0, 'r' );
hold off;
```



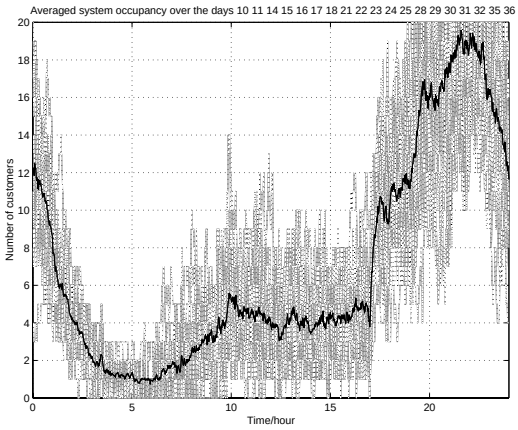
Graph 1
Average number of customers in the staff modem pool on weekdays of January 1998.



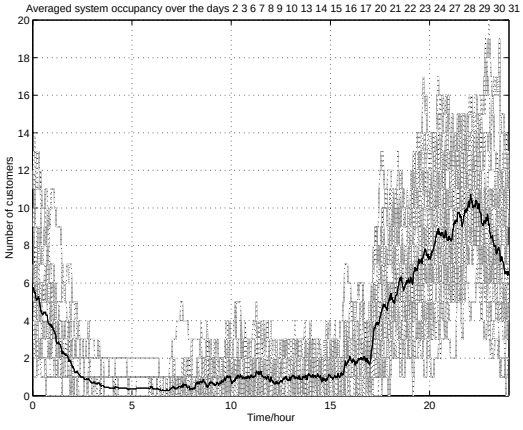
Graph 2
Average number of customers in the staff modem pool on weekdays of May 1998.



Graph 3
Average number of customers in the staff modem pool on weekdays of June 1998.

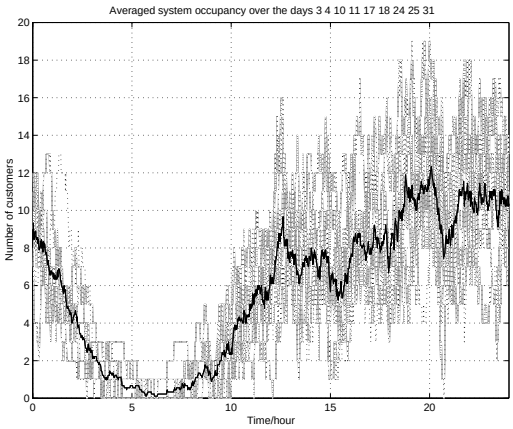


Graph 4
Average number of customers in the staff modem pool on weekdays of September 1998.

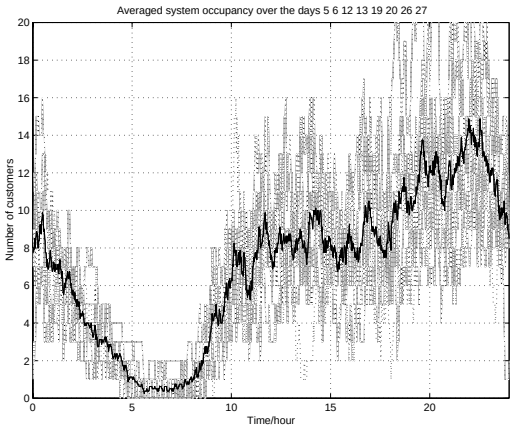


Graph 5
Average number of customers in the staff modem pool on weekdays of October 1997.

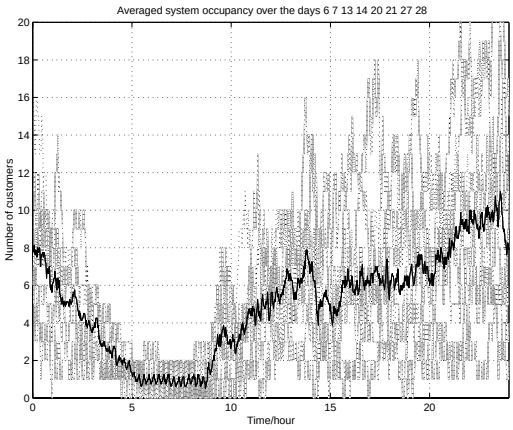
Graphs 1 trough 5 describe the traffic in the staff modem pool on weekdays. There are no significant changes (from October 1997) here. (Graph 5 is reprinted here for comparison. [1]) The number of pool users is slowly increasing, but the use of this pool is still quite weak.



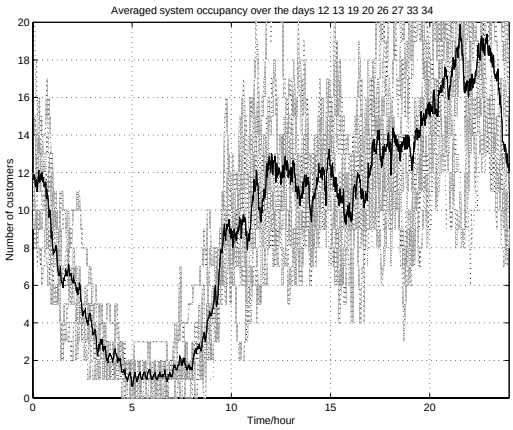
Graph 6
Average number of customers in the staff modem pool on weekends of January 1998.



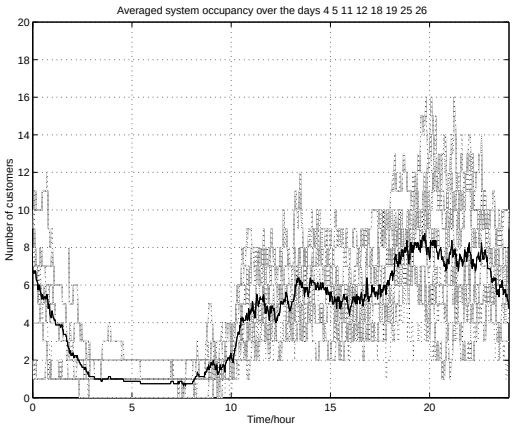
Graph 7
Average number of customers in the staff modem pool on weekends of May 1998.



Graph 8
Average number of customers in the staff modem pool on weekends of June 1998.

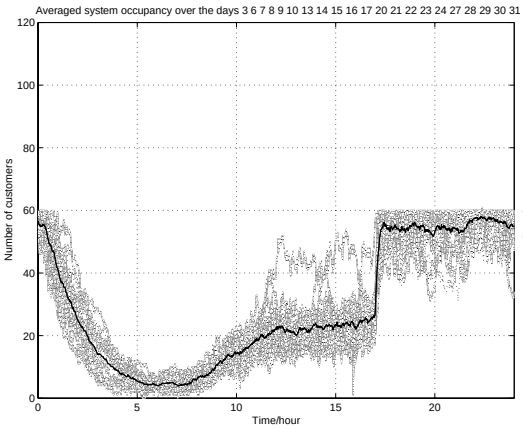


Graph 9
Average number of customers in the staff modem pool on weekends of September 1998.

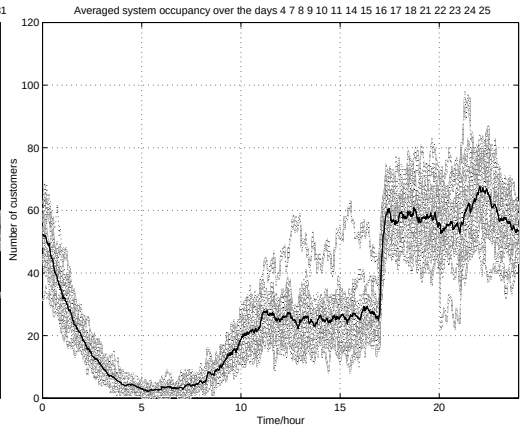


Graph 10
Average number of customers in the staff modem pool on weekends of October 1997.

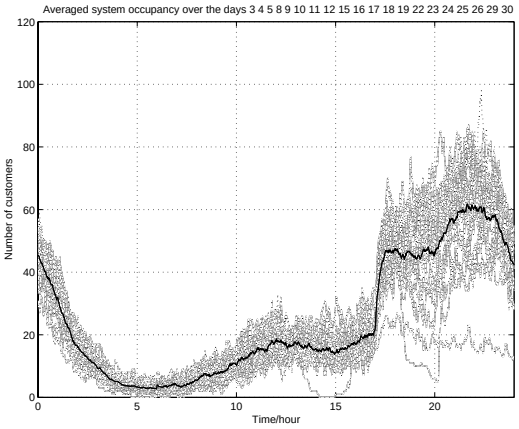
Graphs 6 trough 10 describe the traffic in the staff modem pool on weekends. (Graph 10 is reprinted here for comparison. [1]) The daytime use of this pool seems to be a bit more intense during weekends (which is quite natural, since people are at home).



Graph 11
Average number of customers in the student modem pool on weekdays of January 1998.

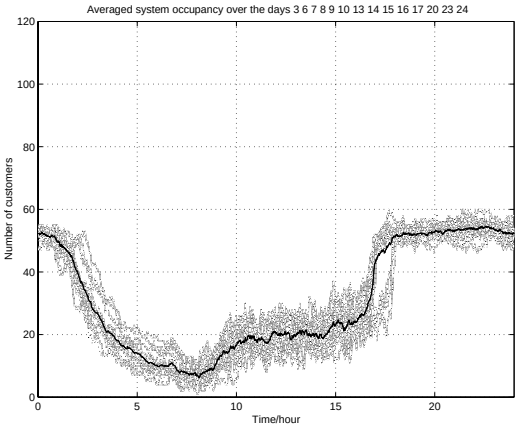


Graph 12
Average number of customers in the student modem pool on weekdays of May 1998.



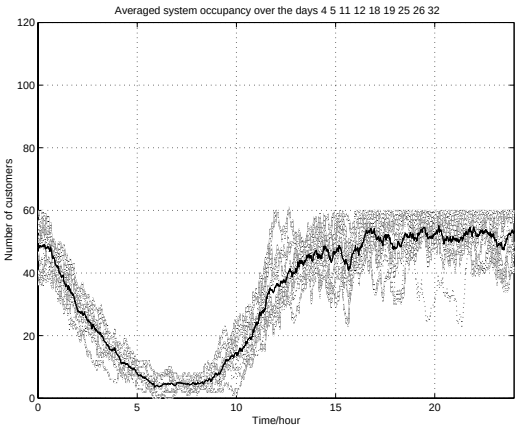
Graph 13
Average number of customers in the student modem pool on weekdays of June 1998.

Graph 14
Average number of customers in the student modem pool on weekdays of September 1998.

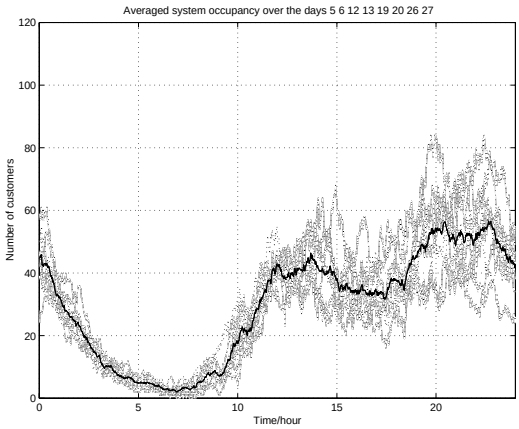


Graph 15
Average number of customers in the student modem pool on weekdays of October 1997.

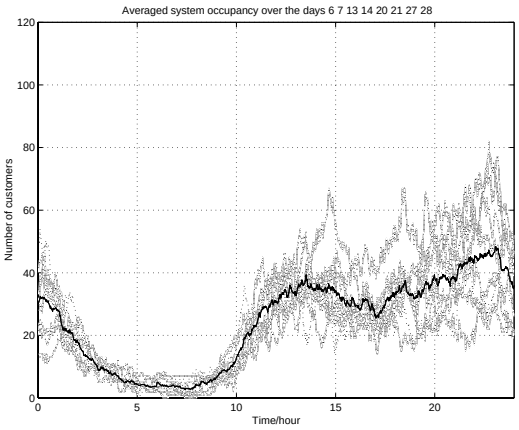
Graphs 11 trough 15 describe the traffic in the student modem pool on weekdays. Traffic in January is still very similar to traffic in October 1997 [1] (graph 15 is reprinted here for comparison), but the next three graphs differ from these. Now there seems to be enough capacity in this modem pool (for 180 users), too. The peak at 17.00 (cheaper rates until 07.00 next morning) is very clear in every graph.



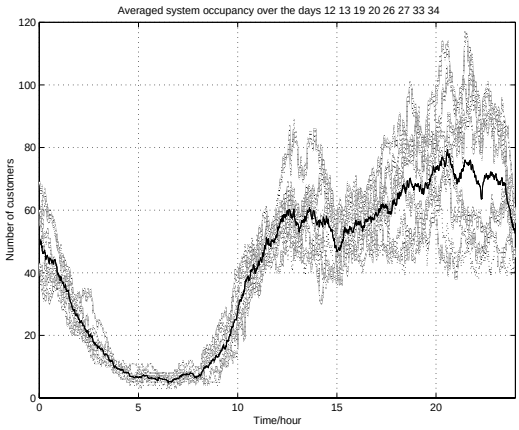
Graph 16
Average number of customers in the student modem pool on weekends of January 1998.



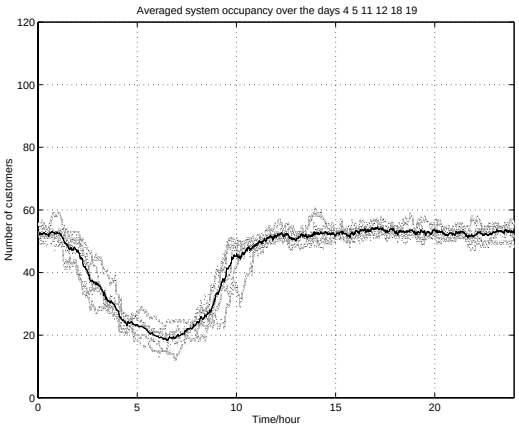
Graph 17
Average number of customers in the student modem pool on weekends of May 1998.



Graph 18
Average number of customers in the student modem pool on weekends of June 1998.



Graph 19
Average number of customers in the student modem pool on weekends of September 1998.



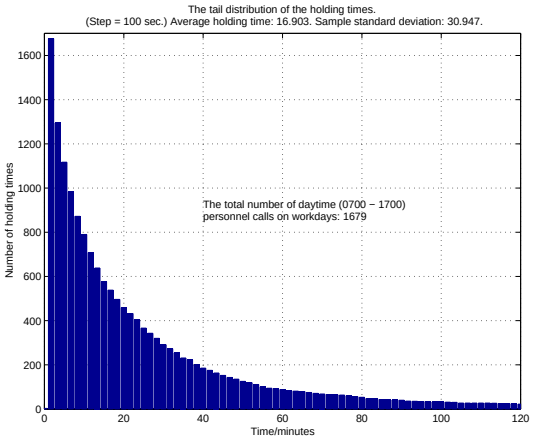
Graph 20
Average number of customers in the student modem pool on weekends of October 1997.

Graphs 16 trough 20 describe the traffic in the student modem pool on weekends. Traffic in January is still very similar to traffic in October 1997 [1] (graph 20 is reprinted here for comparison), but the next three graphs differ from these. Now there seems to be enough capacity in this modem pool (for 180 users), too.

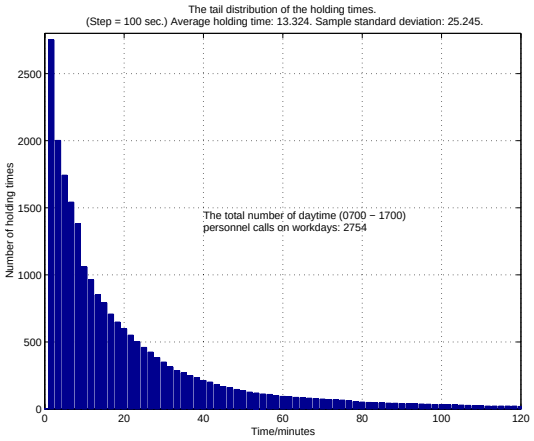
3.1. Holding times

Holding time is the duration of a single connection. Matlab-function *tail_holding_times.m* computes and prints out the tail distribution of the holding times of a particular measurement period. The step for these holding times is set to 100 seconds. The following macro *hold_times_sep_stu_w_d.m* computes the tail distribution for the weekday (07.00 - 17.00) holding times of the student modem pool. It uses the function mentioned earlier.

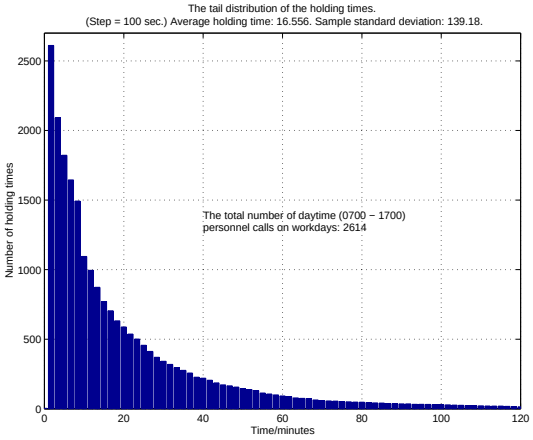
```
load sepstudents.data;

len = length(sepstudents(:,3));
newlen = 0;

for i=1:len,
  if ( ( sepstudents(i,3) ~= [6 0] ) &
     ( ( mod(sepstudents(i,1),86400) < 61200 ) &
       ( mod(sepstudents(i,1),86400) > 25200 ) ) ),
    newlen = (newlen + 1);
  end;
end;

x = zeros(newlen,1);
y = zeros(newlen,1);

k=1;

for j=1:len,
  if ( ( sepstudents(j,3) ~= [6 0] ) &
     ( ( mod(sepstudents(j,1),86400) < 61200 ) &
       ( mod(sepstudents(j,1),86400) > 25200 ) ) ),
    x(k) = sepstudents(j,3);
    y(k) = sepstudents(j,4);
    k = ( k + 1 );
  end;
end;

tail_holding_times( x, y );
axis([0,120,0,15500]);
text(40, 7750, sprintf('The total number of daytime
    (0700 - 1700)\nstudent calls on workdays: %d', newlen));
```



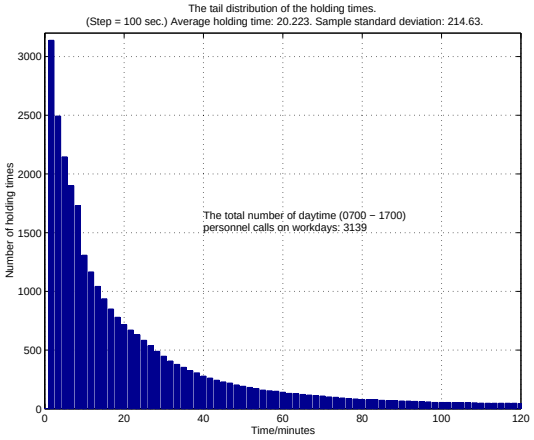
Graph 21
The tail distribution of the holding times in the staff modem pool on weekdays (07.00-17.00) of January 1998.



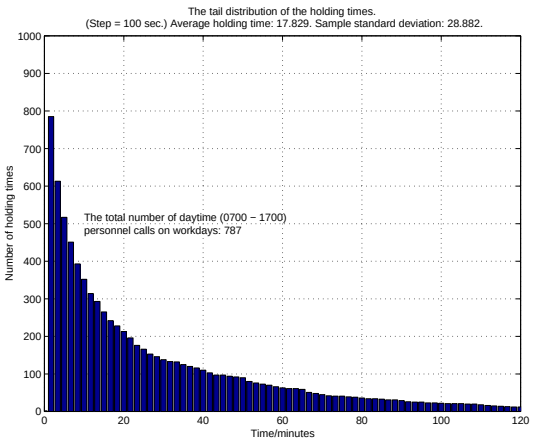
Graph 22
The tail distribution of the holding times in the staff modem pool on weekdays (07.00-17.00) of May 1998.



Graph 23
The tail distribution of the holding times in the staff modem pool on weekdays (07.00-17.00) of June 1998.

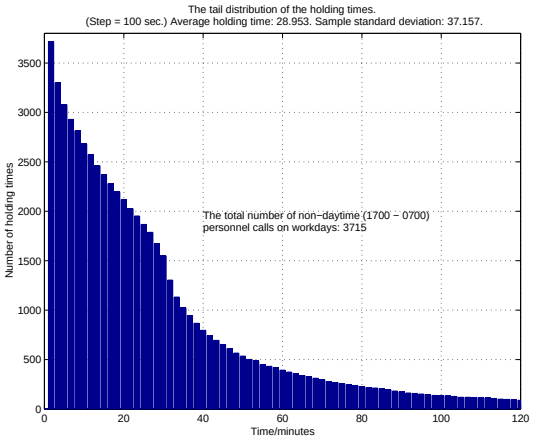


Graph 24
The tail distribution of the holding times in the staff modem pool on weekdays (07.00-17.00) of September 1998.

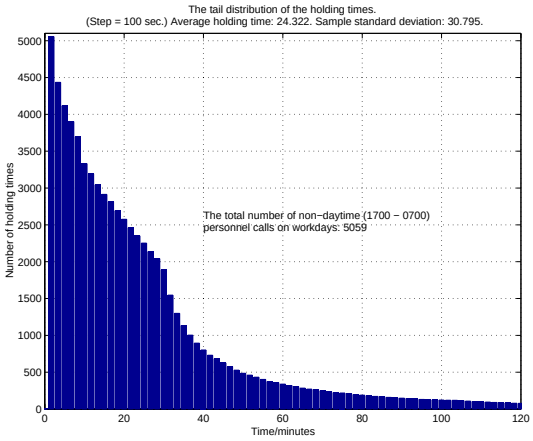


Graph 25
The tail distribution of the holding times in the staff modem pool on weekdays (07.00-17.00) of October 1997.

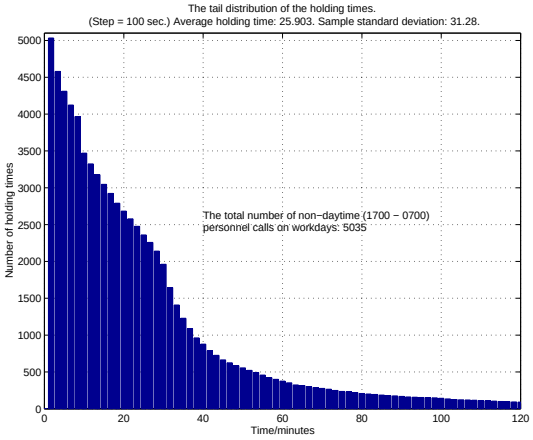
Graphs 21 trough 25 describe the holding times in the staff modem pool on weekdays (07.00 - 17.00). No major changes here. Graph 25 is reprinted here for comparison [1]. Note the different scales in these graphs.



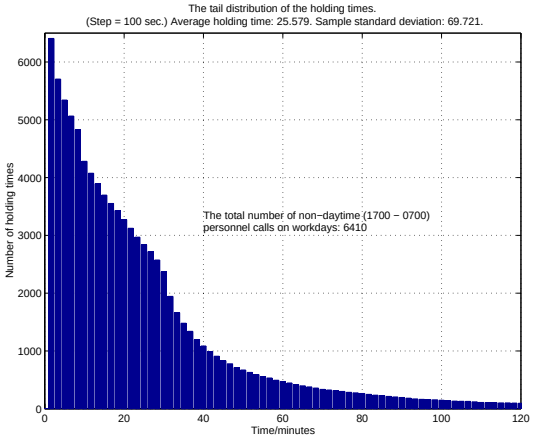
Graph 26
The tail distribution of the holding times in the staff modem pool on weekdays (17.00-07.00) of January 1998.



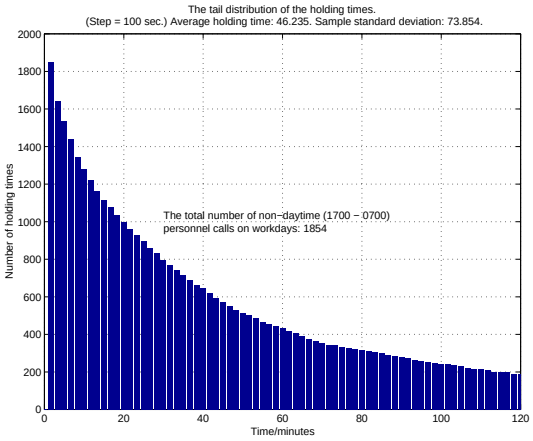
Graph 27
The tail distribution of the holding times in the staff modem pool on weekdays (17.00-07.00) of May 1998.



Graph 28
The tail distribution of the holding times in the staff modem pool on weekdays (17.00-07.00) of June 1998.

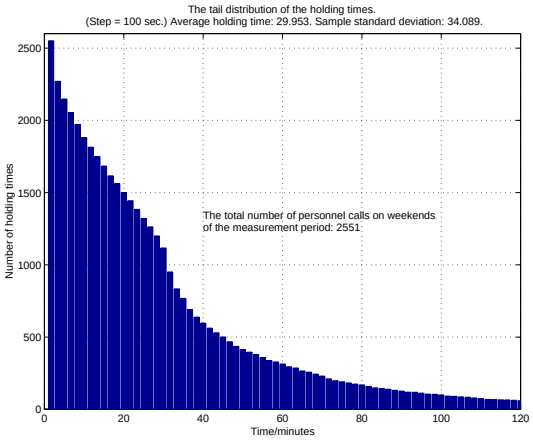


Graph 29
The tail distribution of the holding times in the staff modem pool on weekdays (17.00-07.00) of September 1998.

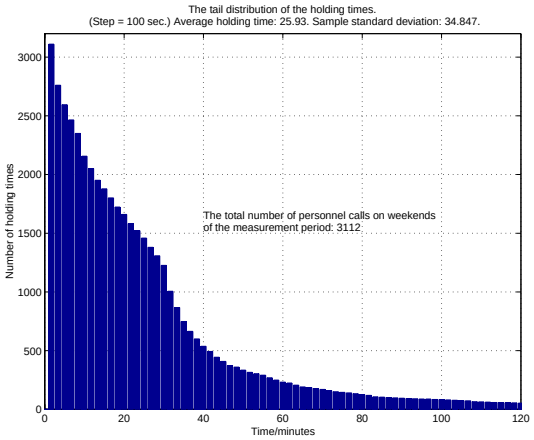


Graph 30
The tail distribution of the holding times in the staff modem pool on weekdays (17.00-07.00) of October 1997.

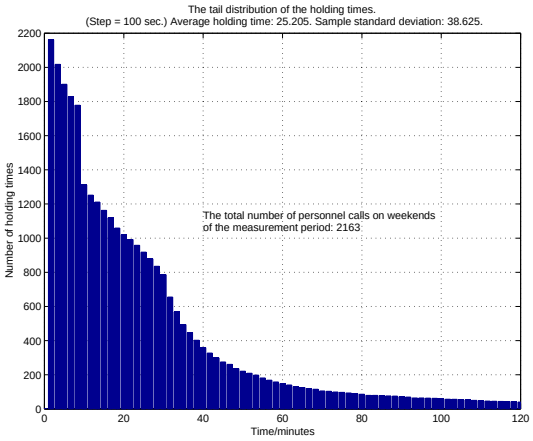
Graphs 26 trough 30 describe the holding times in the staff modem pool on weekdays (17.00 - 07.00). Average holding time has shortened dramatically from October 1997 (46 minutes). The curve seems to be exponential after 30 minutes in graphs 26 - 29. Note the different scales in these graphs.



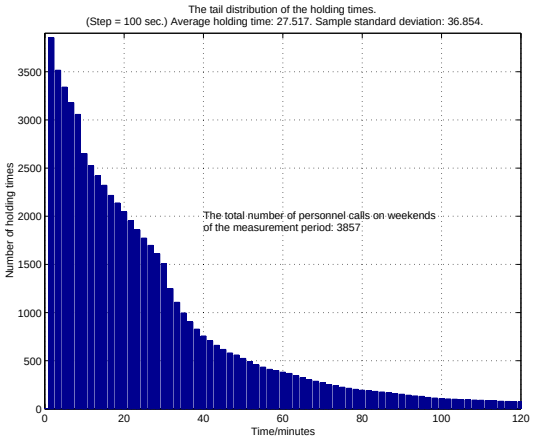
Graph 31
The tail distribution of the holding times in the staff modem pool on weekends of January 1998.



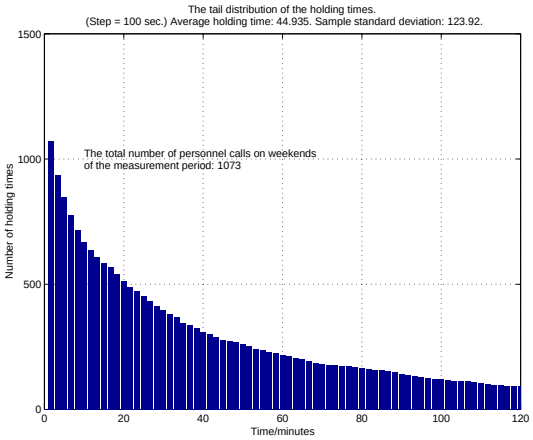
Graph 32
The tail distribution of the holding times in the staff modem pool on weekends of May 1998.



Graph 33
The tail distribution of the holding times in the staff modem pool on weekends of June 1998.

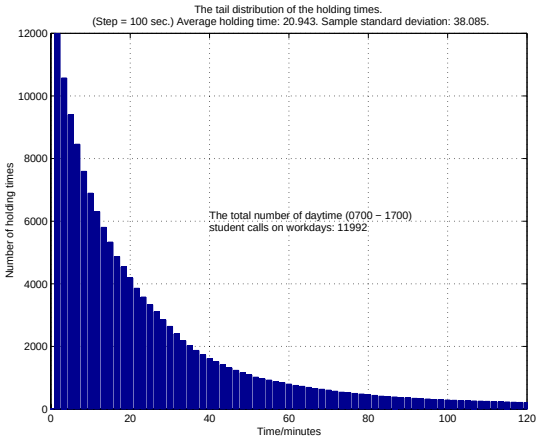


Graph 34
The tail distribution of the holding times in the staff modem pool on weekends of September 1998.

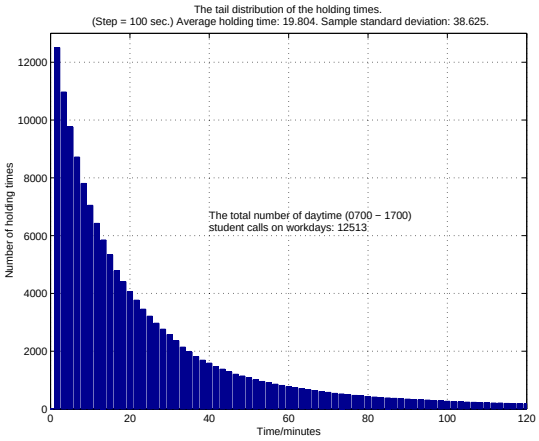


Graph 35
The tail distribution of the holding times in the staff modem pool on weekends of October 1997.

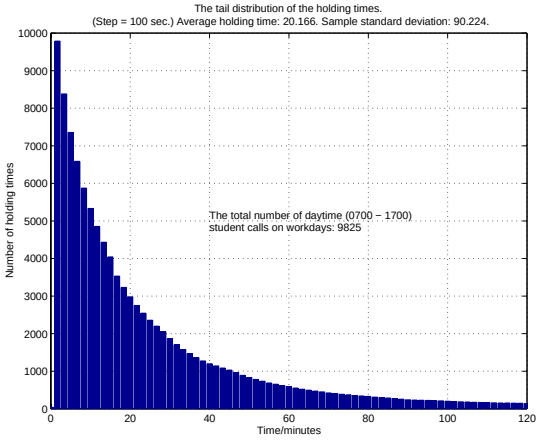
Graphs 31 trough 35 describe the holding times in the staff modem pool on weekends. Average holding time has shortened dramatically from October 1997 (45 minutes). The curve seems to be exponential after 30 minutes in graphs 31 - 34. Note the different scales in these graphs.



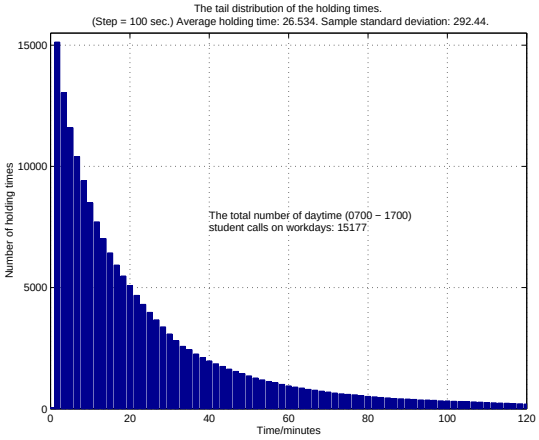
Graph 36
The tail distribution of the holding times in the student modem pool on weekdays (07.00-17.00) of January 1998.



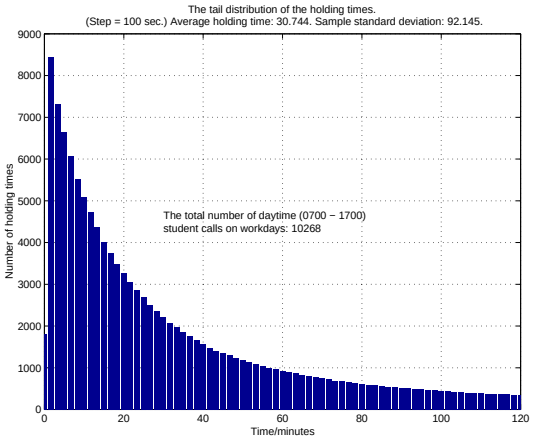
Graph 37
The tail distribution of the holding times in the student modem pool on weekdays (07.00-17.00) of May 1998.



Graph 38
The tail distribution of the holding times in the student modem pool on weekdays (07.00-17.00) of June 1998.

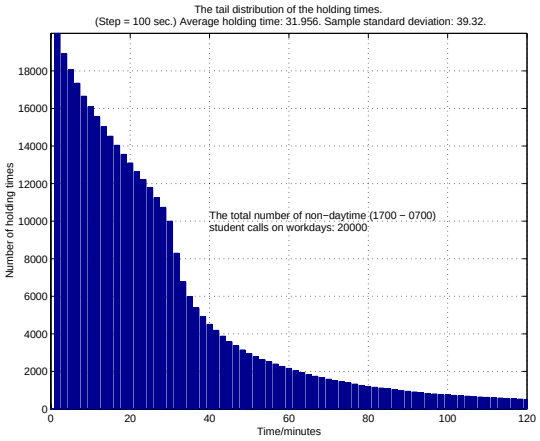


Graph 39
The tail distribution of the holding times in the student modem pool on weekdays (07.00-17.00) of September 1998.

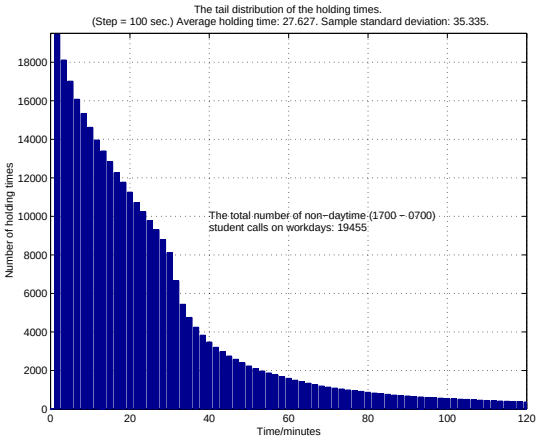


Graph 40
The tail distribution of the holding times in the student modem pool on weekdays (07.00-17.00) of October 1997.

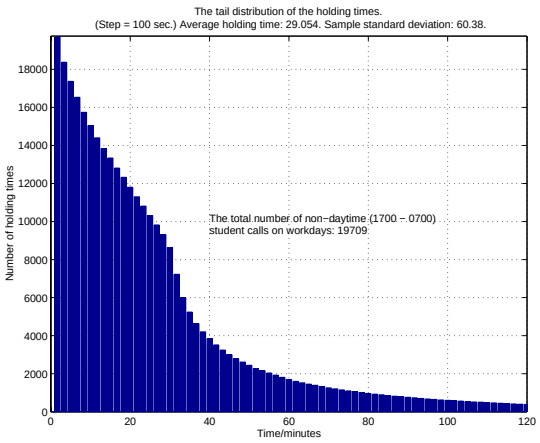
Graphs 36 trough 40 describe the holding times in the student modem pool on weekdays (07.00 - 17.00). No major changes here. Average holding time has shortened a little. Note the different scales in these graphs.



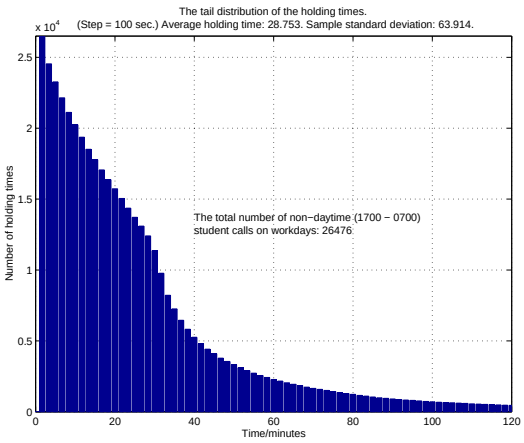
Graph 41
The tail distribution of the holding times in the student modem pool on weekdays (17.00-07.00) of January 1998.



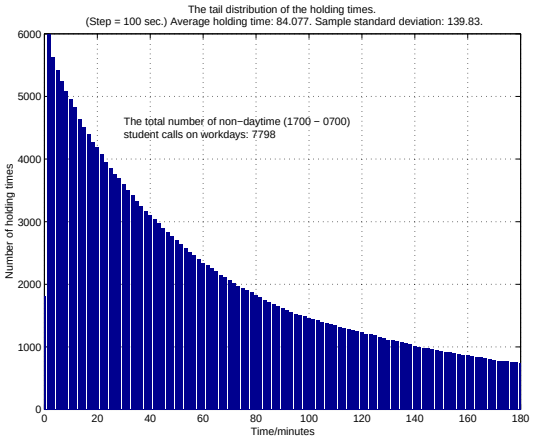
Graph 42
The tail distribution of the holding times in the student modem pool on weekdays (17.00-07.00) of May 1998.



Graph 43
The tail distribution of the holding times in the student modem pool on weekdays (17.00-07.00) of June 1998.

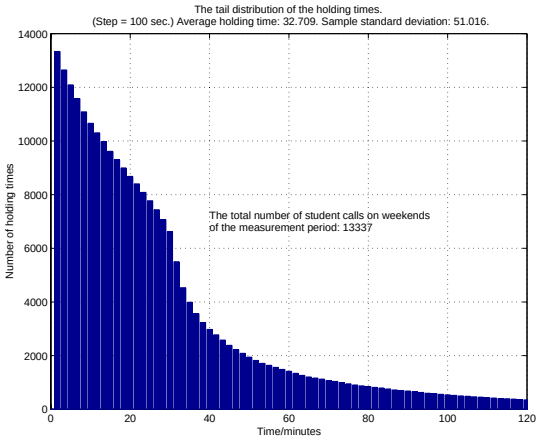


Graph 44
The tail distribution of the holding times in the student modem pool on weekdays (17.00-07.00) of September 1998.

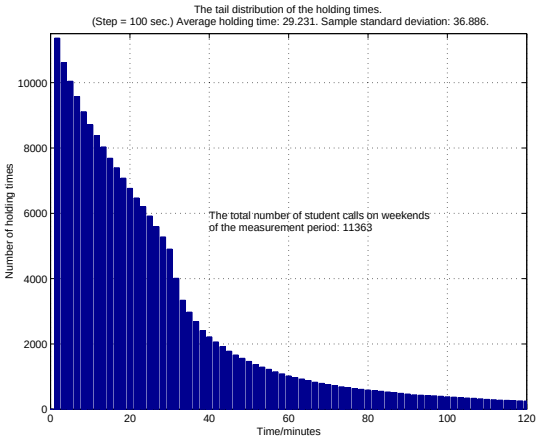


Graph 45
The tail distribution of the holding times in the student modem pool on weekdays (17.00-07.00) of October 1997.

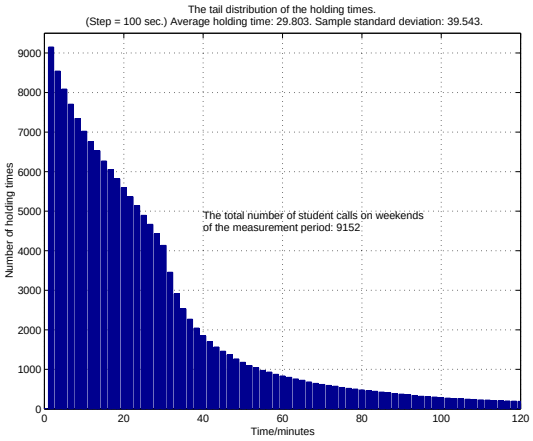
Graphs 41 trough 45 describe the holding times in the student modem pool on weekdays (17.00 - 07.00). Average holding time has dramatically shortened (October 1997: 84 minutes). Sample standard deviation is smaller, too. (October 1997: 140.) Note the different scales in these graphs.



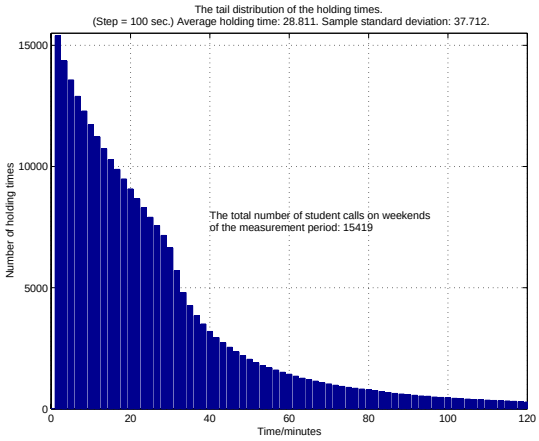
Graph 46
The tail distribution of the holding times in the student modem pool on weekends of January 1998.



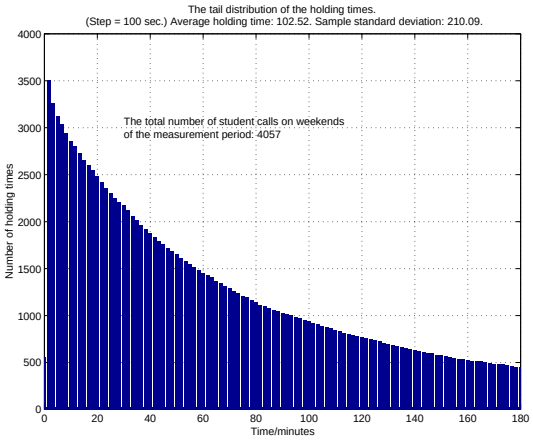
Graph 47
The tail distribution of the holding times in the student modem pool on weekends of May 1998.



Graph 48
The tail distribution of the holding times in the student modem pool on weekends of June 1998.



Graph 49
The tail distribution of the holding times in the student modem pool on weekends of September 1998.



Graph 50
The tail distribution of the holding times in the student modem pool on weekends of October 1997.

Graphs 46 trough 50 describe the holding times in the student modem pool on weekends. Average holding time has dramatically shortened (October 1997: 103 minutes). Sample standard deviation is smaller, too. (October 1997: 210.) Note the different scales in these graphs.

4. Conclusions

It seems that there is no congestion in either one of these modem pools anymore. Staff pool doesn't need all the excess capacity it had before May 1998. The number of simultaneous users in the staff pool hardly never exceeds 20. So, the capacity for 60 users is enough.

The student modem pool was suffering from congestion when it had only 60 lines. Now, with 180 lines, it no longer suffers from congestion. The highest number of simultaneous users this far (in May, June or September - the other months have not been analyzed) has been about 120. There seems to be enough capacity here, too. Together with the increased pool capacity the new call rates (of Helsinki Telephone Corporation) are likely to prevent congestion in the future.

The shortening of the holding times was expected when Helsinki Telephone Corporation announced last winter its new evening and weekend call rates. Surprisingly, this had a great effect on holding times even before the rates were adopted (that was 18.3.1998). The average evening holding time in the student modem pool in January 1998 was only one third of what it was in October 1997 while the pricing policy was exactly the same. Students seem to be very cautious people…

We should notice, that while the holding times have radically shortened (evening and weekend averages in the student pool are now less than 30 minutes), the number of connections per month has also dramatically increased. For example in September 1998 the average evening holding time (in the student pool) was 29 minutes and the number of these connections was 26500, while back in October 1997 [1] these figures were 84 minutes and 8000 customers. It seems, that customers tend to hang up after 30 minutes and then they start a new connection. The use of telephone lines may have decreased a little, because people now make connections only when they have to. Before these new rates it was possible to have a weekend lasting modem connection (or a telephone conversation…) with just 47 p. It might be too expensive (at least for students) now.

5. References

[1] Jani Lakkakorpi: TKK:n modeemipoolien lokitiedostojen analysointi (erikoistyö)
[2] Esa Hyytiä: Modem pools of Helsinki University of Technology (memo)

Appendix I. AWK-scripts

extract_times.awk:

```
#####################################################################
#
#  Simple awk-script to extract starting and ending
#  times of connections from log files.
#
#  Esa Hyytiä 1998
#  Revised by Jani Lakkakorpi 1998

#####################################################################
#
#  This is from Emacs info pages
#

function init_mktime() {
# Initialize table of month lengths
  _tm_months[0,1] = _tm_months[1,1] = 31;
  _tm_months[0,2] = 28; _tm_months[1,2] = 29;
  _tm_months[0,3] = _tm_months[1,3] = 31;
  _tm_months[0,4] = _tm_months[1,4] = 30;
  _tm_months[0,5] = _tm_months[1,5] = 31;
  _tm_months[0,6] = _tm_months[1,6] = 30;
  _tm_months[0,7] = _tm_months[1,7] = 31;
  _tm_months[0,8] = _tm_months[1,8] = 31;
  _tm_months[0,9] = _tm_months[1,9] = 30;
  _tm_months[0,10] = _tm_months[1,10] = 31;
  _tm_months[0,11] = _tm_months[1,11] = 30;
  _tm_months[0,12] = _tm_months[1,12] = 31;
  _tm_debug = 0;
}

# decide if a year is a leap year
function _tm_isleap(year,    ret) {
  ret = (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)
  return ret;
}

# convert a date into seconds
function _tm_addup(a,    total, yearsecs, daysecs, hoursecs, i, j) {
  hoursecs = 60 * 60;
  daysecs = 24 * hoursecs;
  yearsecs = 365 * daysecs;

  total = (a[1] - 1970) * yearsecs;

# extra day for leap years
  for (i = 1970; i < a[1]; i++)
    if (_tm_isleap(i))
      total += daysecs;

  j = _tm_isleap(a[1]);
  for (i = 1; i < a[2]; i++)
    total += _tm_months[j, i] * daysecs;

  total += (a[3] - 1) * daysecs;
  total += a[4] * hoursecs;
  total += a[5] * 60;
  total += a[6];
```

```
  return total;
}

# mktime --- convert a date into seconds,
#            compensate for time zone

function mktime(str,    res1, res2, a, b, i, j, t, diff ) {
  i = 1;
  i = split( str, a, " ");  # dont rely on FS

  if (i != 6)
    return -1;

# force numeric
  for (j in a)
    a[j] += 0;

# validate
  if (a[1] < 1970 ||
      a[2] < 1 || a[2] > 12 ||
      a[3] < 1 || a[3] > 31 ||
      a[4] < 0 || a[4] > 23 ||
      a[5] < 0 || a[5] > 59 ||
      a[6] < 0 || a[6] > 61 )
    return -1;

  res1 = _tm_addup(a);
  t = strftime("%Y %m %d %H %M %S", res1);

  if (_tm_debug)
    printf("(%s) -> (%s)\n", str, t) > "/dev/stderr";

  split(t, b, " ");
  res2 = _tm_addup(b);

  diff = res1 - res2;

  if (_tm_debug)
    printf("diff = %d seconds\n", diff) > "/dev/stderr";

  res1 += diff;

  return res1;
}

##########################################################################

# expects a timestr of format VVKKPPHHMMSS
# and returns integer number of seconds since
# 00:00:00 on January 1, 1970

function convert_time( timestr ) {
  year = substr(timestr,1,2);
  year = year + 1900;
  if ( year < 1970 )
    year = year + 100;

  month = substr(timestr,3,2);
  day   = substr(timestr,5,2);
  hour  = substr(timestr,7,2);
  min   = substr(timestr,9,2);
  sec   = substr(timestr,11,2);

  astr = year " " month " " day " " hour " " min " " sec;
  time = mktime( astr );

  return time
}

##########################################################################
```

```
BEGIN {
#
# First we define the input field separator.
#
  FS = ",";
  init_mktime();

  if (ARGC > 2)
    exitval = 1
  else if (ARGC == 2)
    format = ARGV[1]
}

#
# From each line we get starting and ending points
# and add those moments in events structure
#
{
  secs = convert_time( $1 );
  duration = int( $4 );
  events[ secs ]++;
  end = secs + duration;
  events[ end ]--;
}

#
# All data has been preprocessed and saved in
# events structure.
#
END {

#
#  Find the minimum and maximum point in time
#
  min = -1;
  max = -1;
  for ( time in events) {
    if ( time < min || min == -1 )
      min = time;
    if ( time > max || max == -1 )
      max = time;
  }

  minstr = strftime("%Y %m %d %H:%M:%S", min);
  maxstr = strftime("%Y %m %d %H:%M:%S", max);

#
# Round the starting point to nearest full day
#
  dh = strftime("%H", min);
  dm = strftime("%M", min);
  ds = strftime("%S", min);
  delta = 3600*dh+60*dm+ds;

  printf "Min: %d  =>  %s\n", min, minstr;
  printf "Max: %d  =>  %s\n", max, maxstr;

  newoffset = strftime("%Y %m %d %H:%M:%S", min-delta);
  printf "Newoffset:  %s\n", newoffset;

  for ( time in events) {
    printf "%s\t%d\n", time-min+delta, events[time];
  }
}
```

extract_data.awk:

```
#####################################################################
#
#  Simple awk-script to extract the desired data from log files.
#
#  Esa Hyytid 1998
#  Revised by Jani Lakkakorpi 1998

#####################################################################
#
#  This is from Emacs info pages
#

function init_mktime() {
# Initialize table of month lengths
  _tm_months[0,1] = _tm_months[1,1] = 31;
  _tm_months[0,2] = 28; _tm_months[1,2] = 29;
  _tm_months[0,3] = _tm_months[1,3] = 31;
  _tm_months[0,4] = _tm_months[1,4] = 30;
  _tm_months[0,5] = _tm_months[1,5] = 31;
  _tm_months[0,6] = _tm_months[1,6] = 30;
  _tm_months[0,7] = _tm_months[1,7] = 31;
  _tm_months[0,8] = _tm_months[1,8] = 31;
  _tm_months[0,9] = _tm_months[1,9] = 30;
  _tm_months[0,10] = _tm_months[1,10] = 31;
  _tm_months[0,11] = _tm_months[1,11] = 30;
  _tm_months[0,12] = _tm_months[1,12] = 31;
  _tm_debug = 0;
}

# decide if a year is a leap year
function _tm_isleap(year,    ret) {
  ret = (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)
  return ret;
}

# convert a date into seconds
function _tm_addup(a,    total, yearsecs, daysecs, hoursecs, i, j) {
  hoursecs = 60 * 60;
  daysecs = 24 * hoursecs;
  yearsecs = 365 * daysecs;

  total = (a[1] - 1970) * yearsecs;

# extra day for leap years
  for (i = 1970; i < a[1]; i++)
    if (_tm_isleap(i))
      total += daysecs;

  j = _tm_isleap(a[1]);
  for (i = 1; i < a[2]; i++)
    total += _tm_months[j, i] * daysecs;

  total += (a[3] - 1) * daysecs;
  total += a[4] * hoursecs;
  total += a[5] * 60;
  total += a[6];

  return total;
}

# mktime --- convert a date into seconds,
#            compensate for time zone

function mktime(str,    res1, res2, a, b, i, j, t, diff ) {
  i = 1;
  i = split( str, a, " ");  # dont rely on FS
```

```
  if (i != 6)
    return -1;

# force numeric
  for (j in a)
    a[j] += 0;

# validate
  if (a[1] < 1970 ||
      a[2] < 1 || a[2] > 12 ||
      a[3] < 1 || a[3] > 31 ||
      a[4] < 0 || a[4] > 23 ||
      a[5] < 0 || a[5] > 59 ||
      a[6] < 0 || a[6] > 61 )
    return -1;

  res1 = _tm_addup(a);
  t = strftime("%Y %m %d %H %M %S", res1);

  if (_tm_debug)
    printf("(%s) -> (%s)\n", str, t) > "/dev/stderr";

  split(t, b, " ");
  res2 = _tm_addup(b);

  diff = res1 - res2;

  if (_tm_debug)
    printf("diff = %d seconds\n", diff) > "/dev/stderr";

  res1 += diff;

  return res1;
}

######################################################################

# expects a timestr of format VVKKPPHHMMSS
# and returns integer number of seconds since
# 00:00:00 on January 1, 1970

function convert_time( timestr ) {
  year = substr(timestr,1,2);
  year = year + 1900;
  if ( year < 1970 )
    year = year + 100;

  month = substr(timestr,3,2);
  day   = substr(timestr,5,2);
  hour  = substr(timestr,7,2);
  min   = substr(timestr,9,2);
  sec   = substr(timestr,11,2);

  astr = year " " month " " day " " hour " " min " " sec;
  time = mktime( astr );

  return time
}

######################################################################

BEGIN {
#
# First we define the input field separator.
#
  FS = ",";
  init_mktime();

  if (ARGC > 2)
    exitval = 1
  else if (ARGC == 2)
```

```
    format = ARGV[1]
}

#
# From each line we get starting and ending points
# and add those moments in events structure
#
{
  secs = convert_time( $1 );
  day = int( $2 );
  duration = int( $4 );
  bytes_in = int( $5 );
  bytes_out = int( $6 );
  port_type = int( $8 );
  term_cause = int( $9 );

  events[ secs ]++;
  days[ secs ] = day;
  durations[ secs ] = duration;
  bytes_ins[ secs ] = bytes_in;
  bytes_outs[ secs ] = bytes_out;
  port_types[ secs ] = port_type;
  term_causes[ secs ] = term_cause;
}

#
# All data has been preprocessed and saved in
# events structure.
#
END {

#
#  Find the minimum and maximum point in time
#
  min = -1;
  max = -1;
  for ( time in events) {
    if ( time < min || min == -1 )
      min = time;
    if ( time > max || max == -1 )
      max = time;
  }

  minstr = strftime("%Y %m %d %H:%M:%S", min);
  maxstr = strftime("%Y %m %d %H:%M:%S", max);

#
# Round the starting point to nearest full day
#
  dh = strftime("%H", min);
  dm = strftime("%M", min);
  ds = strftime("%S", min);
  delta = 3600*dh+60*dm+ds;

  printf "Min: %d  =>  %s\n", min, minstr;
  printf "Max: %d  =>  %s\n", max, maxstr;

  newoffset = strftime("%Y %m %d %H:%M:%S", min-delta);
  printf "Newoffset:  %s\n", newoffset;

  for ( time in events) {
    printf "%s\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n", time-min+delta,
                        events[time], days[time],
                        durations[time], bytes_ins[time],
                        bytes_outs[time], port_types[time],
                        term_causes[time];
  }
}
```

Appendix II. Matlab-functions

events.m:

```
function [x,y] = events(xe,ye);
%
% This function plots events. It also returns
% data plotted. X-axis is scaled to days.
%
len = length(xe);
yy = cumsum(ye);
y = zeros(2*len-1,1);
x = zeros(2*len-1,1);
for i=1:(len-1)
  y(2*i-1) = yy(i);
  y(2*i) = yy(i);
  x(2*i-1) = xe(i);
  x(2*i) = xe(i+1);
end;
x(2*len-1) = xe(len);
y(2*len-1) = ye(len);
x = x / 86400;          % Scaled to days
plot(x,y);
grid;
xlabel('Time/days');
ylabel('Number of customers in the system');
```

average.m:

```
function [ax,ay] = average(x, y, days )
%
%  function [ax,ay] = average(x, y, days )
%
%  This function calculates average number of customers
%  in system on given days. Time ax is in hours.
%

% First we make a vector of system occupancy with discrete
% one minute steps

const = 24*60;
x2    = x * const;
xmax  = floor(max( x2 )) + 1;
y2    = zeros( xmax, 1 );

t1 = floor( x(1)*const );
for i=2:length(x)            % first event of that day
    t2 = floor( x(i)*const );
    curr_y = y(i-1);
    while( t1 < t2 )
        y2(t1) = curr_y;
        t1 = t1 + 1;
    end;
end;
y2(t1) = curr_y;

%
% Ok, now collect given days from this "minute" vector
%
```

```
ay = zeros( const, 1 );
ax = (1:const)/60;
for d=1:length( days )
        t1 = (days(d)-1) * const;
        t2 = t1 + const;
        if ( t1 < 1 )
                t1 = 1;
        end;
        if ( t2 > xmax )
                t2 = xmax;
        end;
        while( t1 < t2 )
                tmod = mod(t1,const) + 1;
                ay(tmod) = ay(tmod) + y2(t1);
                t1 = t1 + 1;
        end;
end;
ay = ay / length( days );

plot( ax, ay );
grid;
daystr = sprintf( ' %d', days );
title( sprintf( 'Averaged system occupancy over the days%s', daystr
) );
xlabel( 'Time/hour' );
ylabel( 'Number of customers' );
```

tail_holding_times.m:

```
function [ x, y ] = holding_times( xe, ye );
%
% This function plots the tail distribution of
% the holding times of the connections.
% Vector xe contains the days of the week. Vector ye contains
% the corresponding holding times.
%

ys = sort(ye);                              % First we sort the holding times
len = length(ye);                           % The number of all
connections
points = floor( max(ye) / 100 );            % The number of
observation points
step = 100;                                 % The step of holding
times (100sec)

y = zeros(points+2, 1);
x = zeros(points+2, 1);

x(1) = 0;                                   % Unsuccessful logins?
i=1;

while ( ys(i) == 0 )
  y(1) = ( y(1) + 1 );
  i = ( i + 1 );
end;

start = i;                                  % First non-zero holding
time

left = (len - y(1));

for j=2:(points+2),
  x(j) = ( (j-1)*step );
  y(j) = left;

  while ( (ys(i) > (j-2)*step) & (ys(i) <= ( (j-1)*step )) & (i <
len) ),
    i = ( i + 1 );
```

```
    left = (left - 1);
  end;

end;

dev2 = 0;
ave = sum(ys)/(len-y(1));      % Average does not contain zero
holding times!
for k=start:len,               % Neither does sample standard
deviation...
  dev2 = ( dev2 + 1 / ( (len-y(1)) - 1 ) * ( ys(k) - ave )^2 );
end;
dev = sqrt(dev2);
x = x/60;                      % Scaled to minutes

bar(x, y);
grid;
results = sprintf('Average holding time: %0.5g. Sample standard
deviation: %0.5g.', ave/60,dev/60 );
title( sprintf( 'The tail distribution of the holding times. \n(Step
= 100 sec.) %s', results) );
xlabel('Time/minutes');
ylabel('Number of holding times');
```