Timeplotters:
two tools for visualizing logs and temporal data

Eugene Kirpichov <ekirpichov@gmail.com>

October 30, 2012

Contents

1 Introduction

splot and **timeplot** are tools for visualizing temporal data, especially useful for visualizing data from program logs.

You can get them and see the installation instructions at http://jkff.info/software/timeplotters. Both tools are heavily battle-tested and evolved in response to concrete visualization requirements at a large-scale cluster computing project which the author was part of, though the tools are in no way bound to this particular domain.

timeplot draws quantitative graphs about several streams of events happening over time, e.g. you can use it to compare the distribution of database access latencies from two machines; to draw the number of requests being concurrently processed by each server at each moment, etc.

splot draws a single Gantt-like chart with a birds-eye view of the activity of a number of concurrent processes, color-coding the state of each process at each moment (e.g. processing one of several jobs, or being in a particular stage of processing). This allows to see peculiar system-level behavior patterns and usually allows to instantly isolate system-level performance bottlenecks which very often show themselves as distinct visual patterns. Section Motivation gives an example of what non-trivial aspects of a program's behavior **splot** can show.
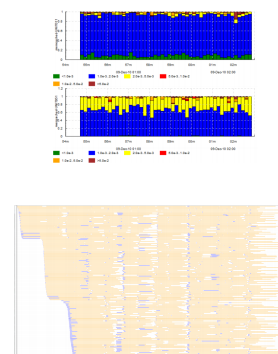
These characteristics make the tools useful for exploratory analysis:

- **Input is trivial to generate** from raw logs by usual text processing tools such as awk or perl

- You can generate **different plots from the same input**

- Fast enough to draw **many millions of events** in tens of seconds, potentially unlimited input size

- Tools are invoked by **one-liners**

Both **timeplot** and **splot** accept input in a strictly specified format, not arbitrary logs. A file in this format is called *trace*. However, this format is designed to be trivial to generate from log files, e.g. using text processing tools such as awk, sed or perl. I use **awk**, it shines at one-liners (google "awk one-liners").

The general pattern of usage is displayed on figure 1: you use an awk one-liner to generate the *trace* and invoke the tool on it.

```
$ awk '/some log entry/{emit an event into trace} \
       /another log entry/{emit another event...}' \
       log.txt > trace.txt
$ splot -if trace.txt -o picture.png ..options..
```

Let us look at a real-life example without considering it in too much detail. Use it only for the purpose of understanding the general pattern of usage.

Log
```
UNIT006 2010-11-13 06:23:27.975 P5872 Info Begin 9a444fde86544c7195
```

One-liner
```
awk `{time=$2 " " $3; core=$1 " " $4} \
    /Begin / {print time " >" core " blue"} \
    /GetCommonData/{print time " >" core " orange"} \
    /End / {print time " <" core}`
```

Trace
```
2010-12-07 13:52:44.738 >UNIT006-P5872 blue
```

One-liner
```
splot -bh 1 -w 1400 -h 800 -expire 10000
```
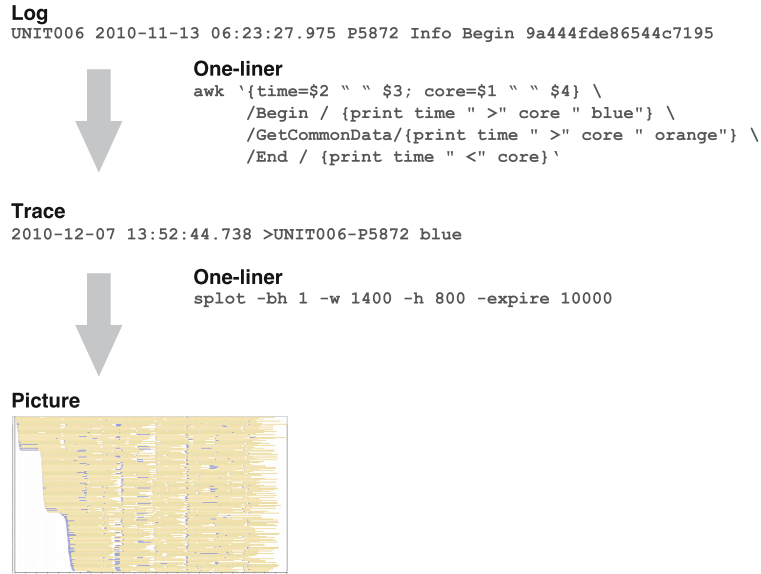
Picture

Figure 1: The general pattern of usage of both **timeplot** and **splot**.

In this example, we're drawing the activity of a computational cluster using **splot**. There's a bunch of worker processes which process tasks from a shared queue. Every task has two stages: 1) fetch data from memcached and 2) run computations. **Here we'll just show how to run the tool, and not explain what the parameters or even the result mean.**

The log entries look like this:

Machine	Date/time	Process ID	Level	Operation	Task ID
UNIT011	2010-12-09 01:54:41.853	P3964	Debug	GetCommonData	390256d1/49

Operation can be one of `Begin` (starting a task), `GetCommonData` (finished getting task data from database, starting computations), `End` (computations for a task finished).

```
$ awk '{time=$2 " " $3; core=$1 " " $4} \
      /Begin /        {print time " >" core " blue"} \
      /GetCommonData/{print time " >" core " orange"} \
      /End /          {print time " <" core}' log.txt > trace.txt
$ splot -if trace.txt -o splot.png -bh 1 -w 1400 -h 800 -expire 10000
```
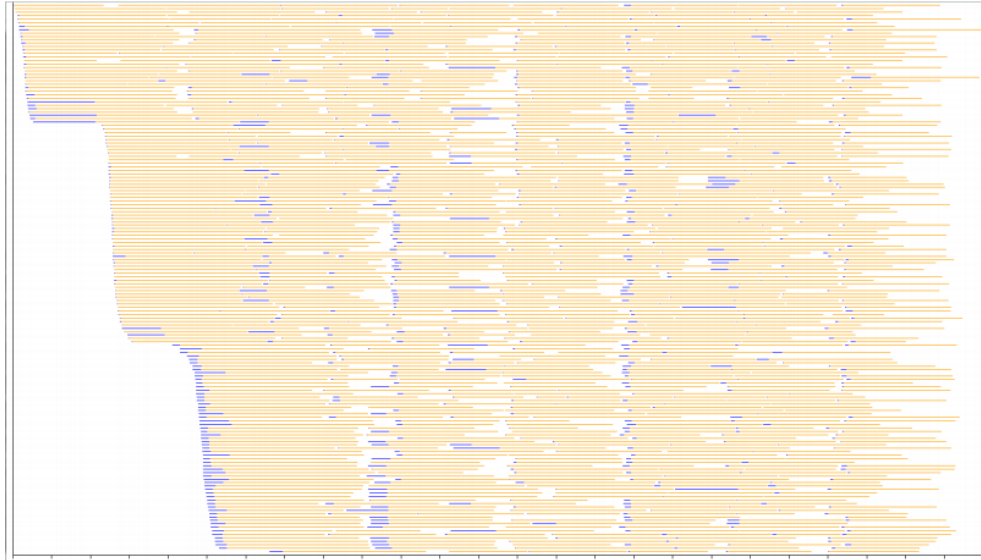
You can see that it took just 2 commands to produce a picture.

The picture is actually very interesting and highlights many performance problems in the original program. The curious reader is advised to look at section Motivation where we discuss this case in detail.

Now let's consider the tools in detail.

2   timeplot — drawing quantitative plots from event streams

timeplot allows you to visualize temporal data in different ways, with the intention to help you spot patterns. It takes as input a sequence of events in a fixed format (there are several types of events), and draws the quantitative characteristics of that sequence in many different ways — e.g. you can **compare the frequency** of different types of events per time unit, or to look at the distribution of **event durations**, etc.

The input to **timeplot** is a sequence of events of the form "event X happened", "interval event X started/finished", "a numeric or discrete variable P took value X". Every event happens on a particular *input track* and at a particular time. The types of events are specified in section Input format. Different types of events can be used with different types of visualizations.

The output is a vertical stack of plots with a common time axis. Each plot corresponds to one *output track*. Usually *input tracks* correspond to output tracks 1-to-1, but sometimes the relation is more complex (multiple tracks on 1 plot, or 1 track on multiple plots) — see Concepts. Every output plot is one of several types (e.g. simple dot plot, line plot, quantile plot, etc.). The kind of plot to draw on a particular output track is specified by the *plot kind mapping*, also described in section Concepts.

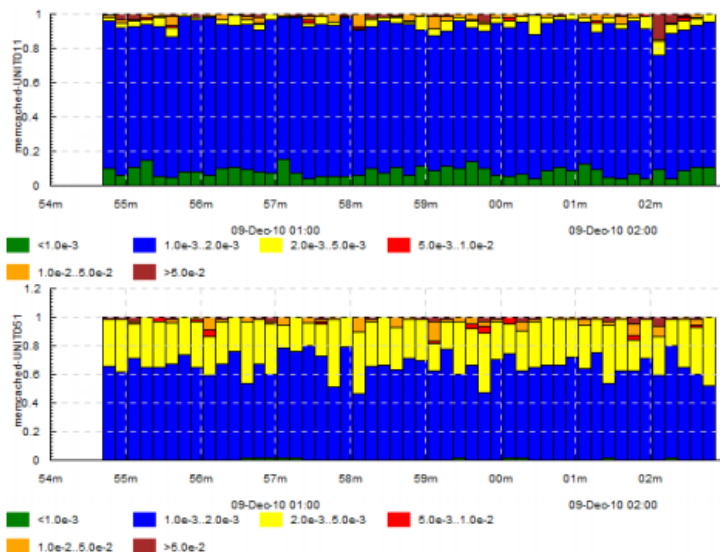An example result of **timeplot** is shown on figure 2.

Figure 2: **timeplot** example: distribution of memcached access latencies from the same rack and from a different rack. Built from a log that had lines about starts and completions of requests to memcached.

2.1 Simple example

This section considers in detail the simplest and most intuitive example possible: a dot plot of a single time-varying value.

Assume that we have a program that sometimes does requests to a database server ("Arcadia" — this is a real-life example from Yandex, and Arcadia is the codename of its core search engine) and logs their durations:

```
...
2010-05-11 18:25:05.435 Arcadia request took 820 ms
...
```

We're interested in looking at the structure of these durations and its change over time. A simple dot plot is a perfect starting point.

First, **generate the trace**: for each line of interest in the log, we need to emit a corresponding *numeric measurement* event into the trace (see reference on event types in section Concepts and on their textual format in section Input format).

The trace should look like this:

```
...
2010-05-11 18:25:05.435 =arc 820
...
```

This is trivial to achieve with an awk one-liner:

```
$ awk '/Arcadia req/{print $1 " " $2 " =arc " $6}' log.txt > trace.txt
```

Now we can **generate the plot**: we need to tell **timeplot** the input and output filenames and what kind of chart to draw.

```
$ tplot -if trace.txt -o arc.png -dk dots
```

The parameter `-dk` means "default chart kind": just draw all tracks (in this case the only track) using this kind of chart (`dots`). So, the track mapping process is involved here in the most trivial form (see section Track and plot kind mapping for less trivial forms).

And we get figure 3 (next page).

We see the following features:

- The times clearly split into higher and lower values, obviously corresponding to cache hits and cache misses within Arcadia.

- At some point the distribution suddenly changes to the worse and requests start taking more time. I do not remember what was the reason for that, but there certainly was one.

- There's a lot of overplotting on the picture; it's difficult to understand the distribution more precisely. To do that, we'll need quantile plots or bin plots (see section Plot kinds). An alternative is to use semi-transparent dots (e.g. `-dk 'dots 0.3'` would give 30% opacity). There exist other ways of coping with overplotting (just google *overplotting*), but they're not currently implemented in **timeplot**.
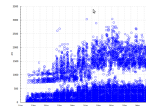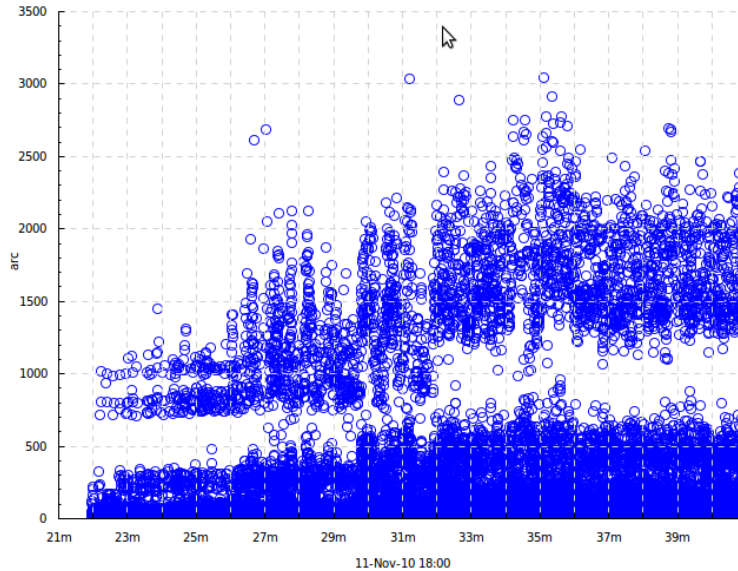
Figure 3: Simple example of **timeplot** — a dot plot of a single variable

Where to go next:

From this simple example you can go several ways:

- Get a glimpse of the power of **timeplot** in a more serious example — continue to section Motivation: More complex example where a reasonably complex real-world example is considered.

- Explore the ways to **map event streams onto charts**: draw multiple tracks, display a single input data point on multiple charts or vice versa, display data points from multiple tracks on a single plot (e.g. a color-coded dot plot) — see section Track and plot kind mapping.

- Learn to use the **more complex event types** (e.g. discrete, impulse, edge and interval events) and to draw other types of charts on them — read section Concepts and continue to section Plot kinds.

- Draw more interesting **types of charts** — just go to section Plot kinds.

- Take a look at the **example charts gallery** and choose something that looks interesting or applicable to your case — go to section Gallery.

2.2   Motivation: More complex example

In this section we'll show a moderately complex real-world example of usage of **timeplot**, with the goal to show its power and inspire the reader to learn more, but without the goal to provide detailed explanations. To become capable of using **timeplot** for similar purposes *yourself*, you'll have to actually read the next chapters.

Consider the log format described in the **timeplot — drawing quantitative plots from event streams —** the one where tasks consist of a "fetching data from memcached" stage and "computational" stage, delimited by `Begin`, `GetCommonData`, `End`.

Suppose that we have several racks of servers and just one memcached server. Let us compare how memcached latencies differ when it is accessed by workers from different racks (since cross-rack access requires an extra network packet hop through a switch device, accesing from the same rack should be faster).

memcached is on rack 1. Let us specifically compare performance on machines `UNIT011` and `UNIT051`. So, we should expect access from `UNIT011` to be faster.

The filtered log looks like this:

```
...
UNIT011 2010-12-09 01:54:41.853 P3964 Debug GetCommonData 390256d1/49
UNIT011 2010-12-09 01:54:41.927 P3964  Info Begin 390256d1/51
UNIT011 2010-12-09 01:54:41.928 P3964 Debug GetCommonData 390256d1/51
UNIT051 2010-12-09 01:54:42.045 P3832  Info Begin 390256d1/99
UNIT051 2010-12-09 01:54:42.045 P3164  Info Begin 390256d1/98
UNIT051 2010-12-09 01:54:42.046 P3164 Debug GetCommonData 390256d1/98
...
```

Let us make this into a trace file for **timeplot**:

```
$ awk '{t=$2 " " $3; p="memcached-" $1 "." $4}
       /Begin /         {print t " >" p}
       /GetCommonData /{print t " <" p}'
       log.txt > trace.txt
```

The trace will look like this:

```
...
2010-12-09 01:54:41.853 <memcached-UNIT011.P3964
2010-12-09 01:54:41.927 >memcached-UNIT011.P3964
2010-12-09 01:54:41.928 <memcached-UNIT011.P3964
2010-12-09 01:54:42.045 >memcached-UNIT051.P3832
2010-12-09 01:54:42.045 >memcached-UNIT051.P3164
2010-12-09 01:54:42.046 <memcached-UNIT051.P3164
...
```

Here the "track names" correspond to different processes (though in general, as we'll see later, track names have broader meaning in **timeplot**), > means the beginning of an activity and < means the end (these are two of the different event types).

Now we'll plot the distribution of durations of memcached access times according to these > and < events.

```
$ tplot -if trace.txt -o latencies.png
        -dk 'within[-] duration binf 10 0.001,0.002,0.005,0.01,0.05'
```

For now do not concern yourself with the meaning of the value of the `-dk` parameter, it will be explained later (in section Plot kinds). Just concentrate on the input (how easy it is to generate from the logs) and the output (how much it tells about the system). The result is shown on figure 4 (next page).
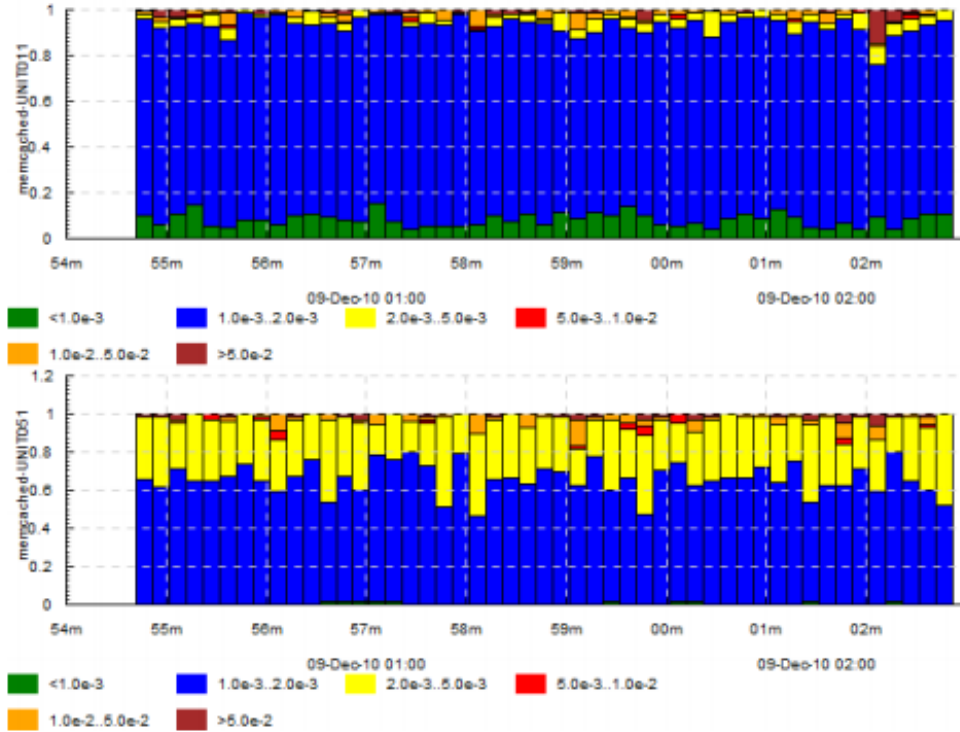

8
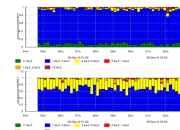
Figure 4: **timeplot** example: distribution of memcached access latencies from the same rack and from a different rack.

Explanation of the output: The graph above corresponds to access from UNIT011, below from UNIT051. Both graphs have time on the X axis and latency on the Y axis. Time is cut into 10-second bins represented by a stack of colored bars. Within each stack (see legend and compare to the invocation of tplot above):

- Height of the green bar shows the fraction of latencies under 0.001s

- Height of the blue bar shows the fraction of latencies in 0.001s–0.002s

- Height of the yellow bar shows the fraction of latencies in 0.002s–0.005s

- Height of the red bar shows the fraction of latencies in 0.005s–0.01s

- Height of the orange bar shows the fraction of latencies in 0.01s–0.05s

- Height of the brown bar shows the fraction of latencies above 0.05s

Together these fractions add up to 1.

The graphs differ:

- There are **no green bars** on the second graph, i.e. access from a different rack is never under 0.001s

- The **yellow bars are a lot larger** on the second graph, i.e. times in 0.002s–0.005s are much more frequent when accessing from a different rack

To reiterate: given the log, the following commands:

9

```
$ awk '{t=$2 " " $3; p="memcached-" $1 "." $4}
       /Begin /         {print t " >" p}
       /GetCommonData /{print t " <" p}'
       log.txt > trace.txt
$ tplot -if trace.txt -o latencies.png
        -dk 'within[-] duration binf 10 0.001,0.002,0.005,0.01,0.05'
```

. . . give us figure 4 which shows how exactly the distributions of access latencies from different racks differ and emphasize the importance of choosing a nearby memcached according to network topology.

This example illustrated the mode of usage of **timeplot**, the ease of generating input for it from an arbitrary log and the terseness of its syntax for specifying the kind of graph to be plotted. We'll now give some basic definitions and then proceed to a formal and exhaustive reference.

2.3   Concepts

This section lists the concepts necessary for precisely understanding the rest of the manual. You can quickly skim over them now just to get a feeling of what they're about, and return to them later when something is unclear.

Event The atomic unit of information in the input trace. It can be one of several types: "something has happened" (this is called an *impulse event*), "something has started/finished" (this is called an *edge event*, and the activity delimited by start/finish is called an *interval event*), "some magnitude had a particular value" (*measurement event*) etc. The types of events correspond to what is usually found in typical program log entries, so they're very easy to generate from logs. Every event happens on a particular *input track* and at a particular time, for example: `2012-06-04 14:24:05.384 =rtime.mcd1 5.371`, this is a numeric measurement event ("="), here `rtime.mcd1` is the input track name and `2012-06-04 14:24:05.384` is the timestamp.

Input track A named group of events in the *input trace*. Usually corresponds to a single magnitude being measured or to a single family of activities, e.g. there could be an input track for request execution times named "rtime" or a track for types of received messages by client C1 named "mtype-C1". Thus, an input track nearly always consists of events of the same type. Here we would have the input trace consist of numeric measurement events with track "rtime" (see different types of events described in section Input format).

Output track A named group of events in the *output plots*. The output track of an event is often equal to its input track, but in general it is determined from its input track by the process of *track mapping* described in section Track and plot kind mapping. All events with the same output track are shown on the same output plot.

Output plot A single plot in the resulting picture. The picture consists of several output plots vertically stacked together with a common time axis. A single output plot is based on values from a single output track, which may have events from one or more input tracks.

Track mapping The process by which events from different input tracks are mapped onto output tracks, e.g. to make events from multiple input tracks participate in a single output plot, or to make events from a single input track participate in multiple output plots. It is controlled, together with *plot kind mapping*, by `+k`, `-k`, `+dk`, `-dk` options and by the `within[SEP]` plot kind. The process is described in section Track and plot kind mapping.

Plot kind The type of an output plot: e.g. dot plot, line plot, quantile plot etc. There are also a couple of "meta" plot kinds: duration plots and "within"-plots. Plot kinds usually have parameters, e.g. the percentiles of interest on a quantile plot. Plot kinds are described in section Plot kinds.

Plot kind mapping The process by which we determine what plot kind to use for visualizing a particular output track. It also depends on +k, -k, +dk and -dk options, happens together with *track mapping* and is described in detail in section Track and plot kind mapping.

To put it together: Input events belong to input tracks and get mapped onto output tracks via track mapping. Every output track gives rise to an output plot, its kind determined by plot kind mapping. Output plots are stacked vertically with a common time axis.

The following concepts are important for understanding the different event types and some plots produced from them:

Measurement event An event that denotes that a particular parameter was measured to have a particular value (e.g.: the amount of free memory was measured to be 2.5Gb), or that something happened with a particular value of a parameter. E.g.: an I/O write request *for 65536 bytes* arrived (e.g. `... =writeBytes 65536`) — a numeric measurement; or an I/O request *of type "write"* has arrived (e.g. `... =requestType 'WRITE`) — a discrete measurement.

Impulse event An input event without parameters that just denotes that something has happened, determined solely by its input track. E.g. if you're interested in the number of completed requests per second (e.g. `... !requestCompleted`), you can have an input trace with impulse events on the track `requestCompleted` and draw an "activity count" plot of that (**??**).

Edge event (counter bump) An input event without parameters that denotes that some activity (interval event) has *started* or *finished*. It can at the same time be thought of as a bump of +1 (`>request`) or -1 (`<request`) of the logical counter associated with this event's input track.

Counter A logical time-varying variable associated with an *input track* which can be bumped by start/finish (*edge*) events. E.g. if your input trace includes events like "started/finished executing a request" (e.g. `... >request, <request`), then **timeplot** will keep a logical counter that can be used to plot the number of concurrently executing requests.

Interval event The logical activity delimited by a start and finish event. More precisely, the period during which a *counter* is greater than zero. The duration of interval events can be measured (producing a bunch of numeric measurement events) and you can draw all kinds of plots about these numeric events, e.g. if your input trace has "request started/request finished" events but doesn't have numeric events about request durations, you can still draw a quantile plot of request durations. This is called a *duration plot*.

Duration plot A meta-plot which means "plot something else, using as input the durations of interval events formed by edge events of this track" (e.g. `duration quantile 1 0.5,0.75,0.95`).

We advise you to revisit section Motivation: More complex example and see if you now better understand the concepts involved there.

2.4 Input format

The general format of a **timeplot** event is as follows:

```
TIMESTAMP [!<>=@]TRACK [VALUE]
```

For example:

```
2012-06-04 14:24:13.389 !requestCompleted
2012-06-04 14:24:13.389 !userLogIn Joe
2012-06-04 14:24:13.389 >request.mcd1
2012-06-04 14:24:13.389 <request.mcd1
2012-06-04 14:24:13.389 =rtime 37.2
2012-06-04 14:24:13.389 =cache 'MISS
2012-06-04 14:24:13.389 @phase blue
```

The table below explains the meaning of all the event types.

Syntax	Meaning	Example
`!TRACK`	Impulse event.	`!requestCompleted`
`!TRACK TEXT`	Impulse event with a text label.	`!userLogIn Joe`
`>TRACK`	Interval event start / counter bump +1.	`>requestCompleted`
`<TRACK`	Interval event finish / counter bump -1.	`<requestCompleted`
`=TRACK NUMBER`	Numeric measurement event.	`=rtime 37.2`
`=TRACK 'TEXT`	Discrete measurement event.	`=cache 'MISS`
`@TRACK COLOR`	Colored interval event start.	`@phase blue`

2.5 Plot kinds

This section describes all the chart kinds supported by **timeplot** and gives recommendations on their usage.

Remember that the input data for each chart is events with a common *output track* (see section Track and plot kind mapping), i.e. possibly events from multiple *input tracks*.

Many of the chart kinds accept a "bin width" parameter: for example, `quantile 10 0.5,0.9,0.95` has a bin width of 10 seconds. This means that **timeplot** will slice the time axis into 10-second bins, compute the 50%, 90% and 95% quantiles of data in each bin and visualize the result in some way.

2.5.1 Special kinds

Empty chart — `none`. This means "do not draw this output track at all". This is useful if you have prepared a trace from a large log file and invoke **timeplot** several times on it, omitting some tracks altogether.

Chart over interval durations — `duration XXX` or `duration drop XXX`. This means "draw chart of kind XXX over the numeric durations of interval events delimited by start/finish events (">"/"<") on this track" (see section Concepts). Durations are measured in seconds. If `drop` is specified, then names of the original input tracks are replaced by the name of the output track (of course, this only makes a difference if multiple input tracks map to this output track).
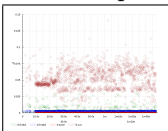
For example, suppose you're measuring **processing of requests by several stages** of a single-threaded pipeline (i.e., every stage of the pipeline processes at most 1 item at a time). Then your log might say:

```
.... >process.stage1
.... >process.stage2
.... <process.stage1
.... >process.stage3
.... >process.stage1
.... <process.stage2
....
```

If you're interested in making a dot plot of processing durations by different stages of the pipeline, you might want to do it in two ways:

- 1 plot per stage: just use `-dk 'duration dots'`.

- 1 plot combined, with dots colored according to stage: use `-dk 'within[.] duration dots'`. Then all these tracks will be put onto the same otput track `process`, but their original names will be preserved

  and you'll get a plot like this: .

- If you use `-dk 'within[.] duration drop dots'`, you'll get this: , but this is likely
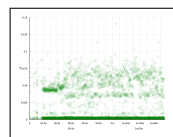
  not what you wanted.

On the other hand, suppose you're measuring the durations of **processing of requests themselves**. Then your log might say:

13

```
.... >process.14ca3ef7
.... <process.a28f3b13
....
```

In this hypothetical log, all request ids are unique (contrary to pipeline stage names in the previous example). So, if we try drawing a dot plot of their durations using `-dk 'within[.] duration dots'`, we'll get a plot where every dot has a different color, which makes no sense (and is not even processible at all by plot types such as `quantile`). That's what `duration drop XXX` is for: we can use `-dk 'within[.] duration drop dots'` and the output track `process` will contain durations of the requests with input track names not like `process.14ca3ef7`, but simply `process`, drawn in a single color.

Chart for M:1 track mapping — `within`. This is not a chart kind proper, it's a means for displaying values from multiple input tracks on a single chart. See section Track and plot kind mapping.

2.5.2 Kinds for numeric data

All the following chart kinds work on numeric measurement events, i.e. events of the form `.... =TRACK VALUE`, e.g. `2012-08-07 18:04:35 =rtime 52.1`.

Simple dot plot — `dots [ALPHA]`. When multiple input tracks map to this output track, the different input tracks are drawn with different colors. `ALPHA` is an optional opaqueness level: 0 means completely transparent, 1 means completely opaque. This helps to deal with overplotting, when there are too many values to display.

Example with 1 input track (`dots` without alpha):
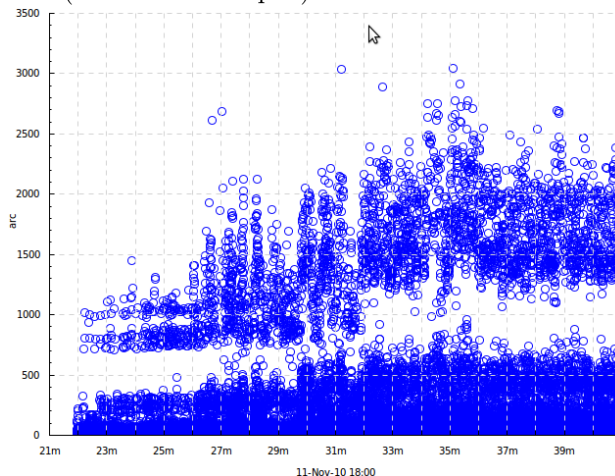
This is the same graph as in section Simple example: response times of a search engine server are drawn. Example with multiple input tracks (`dots` without alpha):
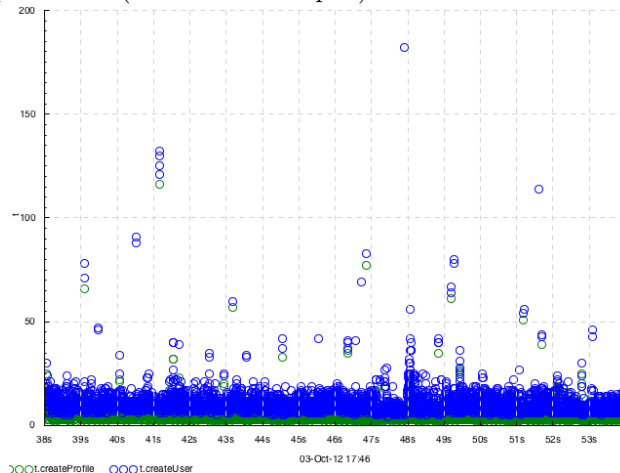
Previous example with `dots 0.2`:

200

150

100

50

0

38s 39s 40s 41s 42s 43s 44s 45s 46s 47s 48s 49s 50s 51s 52s 53s

03-Oct-12 17:46

t.createProfile   t.createUser
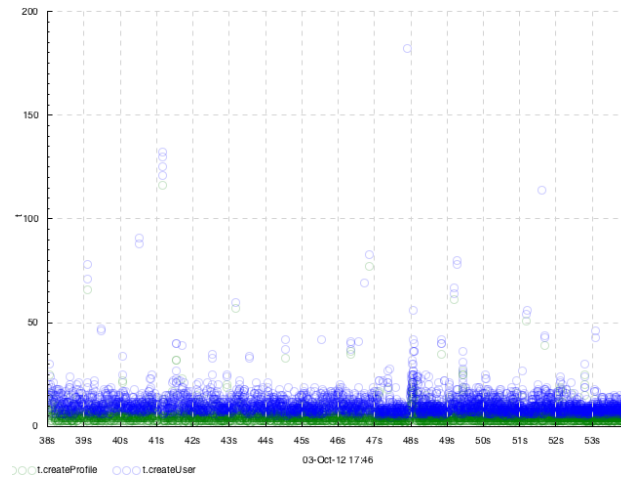
Simple connected line plot — `lines`. Same as `dots`, but the dots are connected. When multiple input tracks map to this output track, the different input tracks are drawn with different colors.

Example with 1 input track per output track:
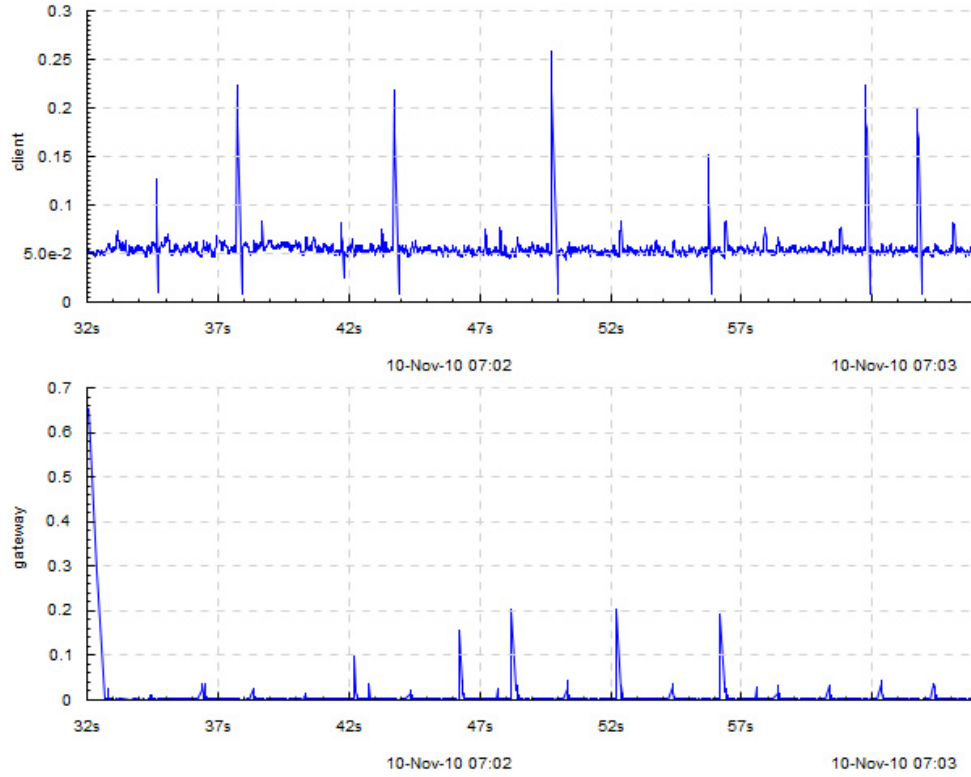
These are request execution times as seen by 1) the caller (*client*) and 2) the callee (*gateway*), plotted from a trace with events of the form `=client 0.053`.

Line plot of binned sum — `sum N [TYPE]` — slice time axis into N-second bins; draw a simple line plot of sums of values in each bin. When multiple input tracks map to this output track, the different input tracks are drawn with different colors. If TYPE is `overlayed`, subplots for different input tracks are just overlayed on each other. If TYPE is `stacked` (default), they are "accumulated" so you can see how the total sum adds up from them. This is useful to understand, e.g., how big is the role of a step of some computation in the total time it takes. If the number of values per bin is usually the same, then `sum` also gives an impression of the average value[1].

Example with two tracks, on the same data as in `dots: sum 1 overlayed`
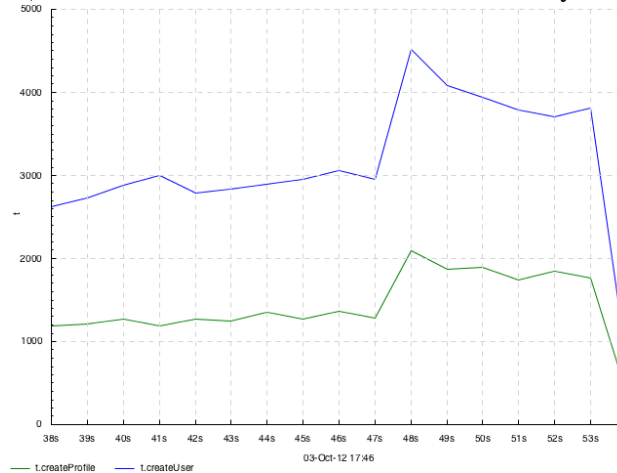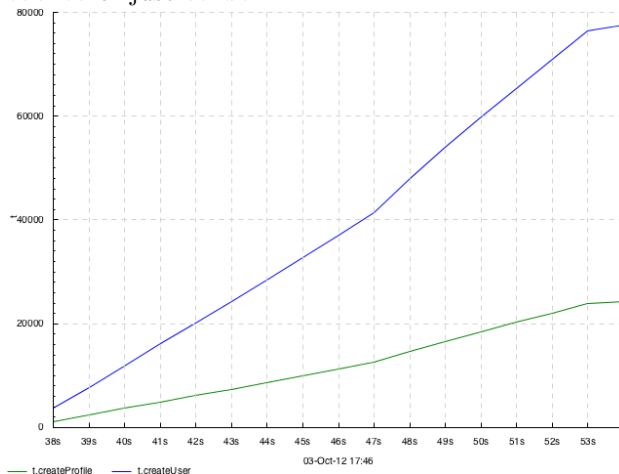

Same with `sum 1 stacked` or just `sum 1`:

---

[1] It's probably worth implementing an actual `avg` chart kind.

Line plot of cumulative sum — `cumsum N [TYPE]` — same as `sum`, but accumulation happens not in every bin but from the beginning of time. Example with two tracks, on the same data as in `dots`: `cumsum 1 overlayed`
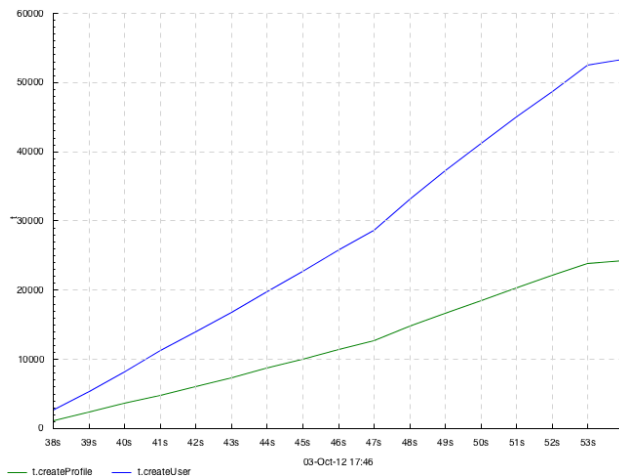

Same with `cumsum 1 stacked` or just `cumsum 1`:

Quantile plot — `quantile N` $q_1, q_2, .., q_M$ — slice time axis into N-second bins; draw a bar chart of q_1'th, q_2'th etc. percentiles of numeric values in each bin. All of q_1, q_2... are numbers between 0 and 1. For example, 0.75 means the 75% quantile, i.e. the value x such that 75% of the values in this bin are smaller than x. Obviously, the 0'th quantile is the minimum value in the bin, and the 1'th quantile is the maximum value. 0 and 1 are always implicitly added to the quantiles you specify.

Specifically, within each bin, stacked bars of different color are drawn: $0..q_1$, $q_1..q_2$, ..., $q_M..1$. This means that the lower edge of what you see is the minimum value within that bin, and the upper edge is the maximum value; and bar boundaries correspond to the quantiles you asked for.

Example:
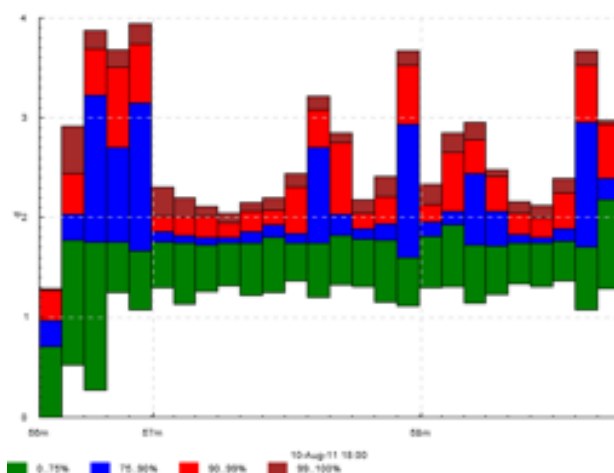
This is a chart of kind `quantile 60 0.75,0.90,0.99`. The green bar spans from minimum in a bin to the 75% quantile, the blue bar starts at 75% and ends at 90%, the red bar starts at 90% and ends at 99% and the brown bar starts at 99% and ends at maximum. All this is reflected in the legend.

Another example:
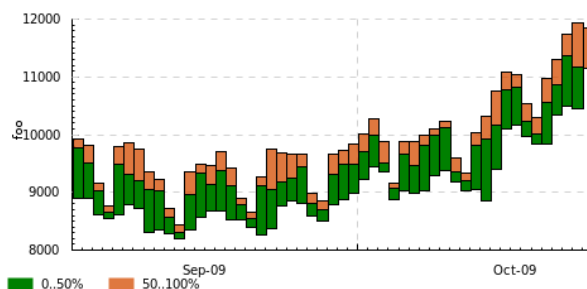
This was drawn with kind `quantile 86400 0.5`. The green bar spans from day minimum to median and the brown bar spans from median to day maximum.

Bin frequency plot — `binf N` $v_1, v_2, .., v_M$ — slice time axis into N-second bins; draw a bar chart of frequencies of values falling into the bins $< v_1$, $v_1..v_2$, ..., $> v_M$. Frequencies are numbers from 0 to 1, so they add up to 1, so the total height of the bars is always 1. **Bin histogram plot** — `binh N` $v_1, v_2, .., v_M$ — same as `binf`, but instead of frequencies, the absolute number of occurences of each bin is drawn.

Example of both `binf` and `binh`:
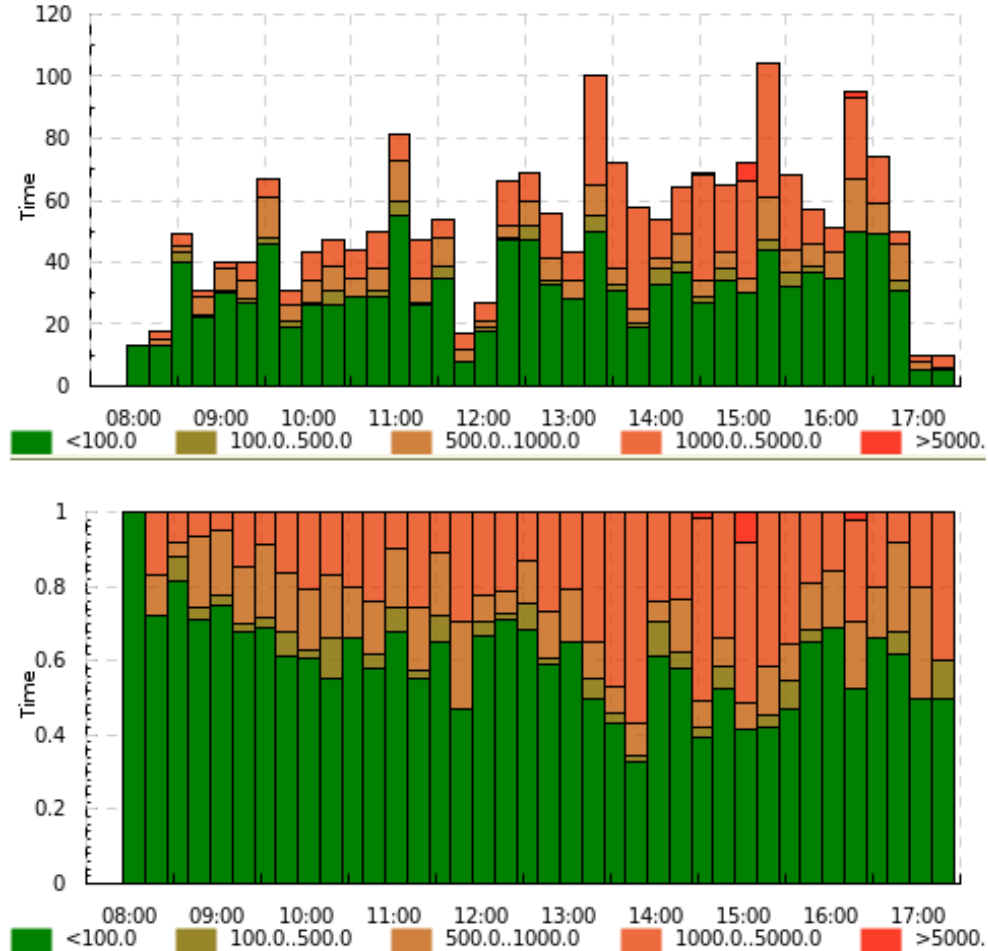
*(this graph was made with an old version of **timeplot** which had a pretty ugly color scheme...)*

Here the same value (page download time by a web crawler) is drawn using both `binf` and `binh`: we see how frequently the download took below 100ms, 100 to 500ms, 500 to 1000ms etc. On the top graph we see *what number* of pages took that much to download, and on the bottom graph we see *what fraction* of pages took that much. In different situations both of these can be useful.

2.5.3 Kinds for counters

Event plot — `event` — draw intervals delimited by `>` and `<` input events and markers according to `@` and `!` events. In the more complex case where `>` and `<` do not strictly alternate, they are interpreted as "+1" and "-1"-s to a counter and **timeplot** draws intervals when the counter is greater than zero. Multiple input tracks per output track are not supported. If you need to draw something similar to what `event` does, but more complex, you might consider using **splot** instead. `@TRACK COLOR` events act as `>` but the bar will have the indicated color. `!TRACK` will draw a vertical red dash and `!TRACK TEXT` will draw a red dash with a text label.

Example:
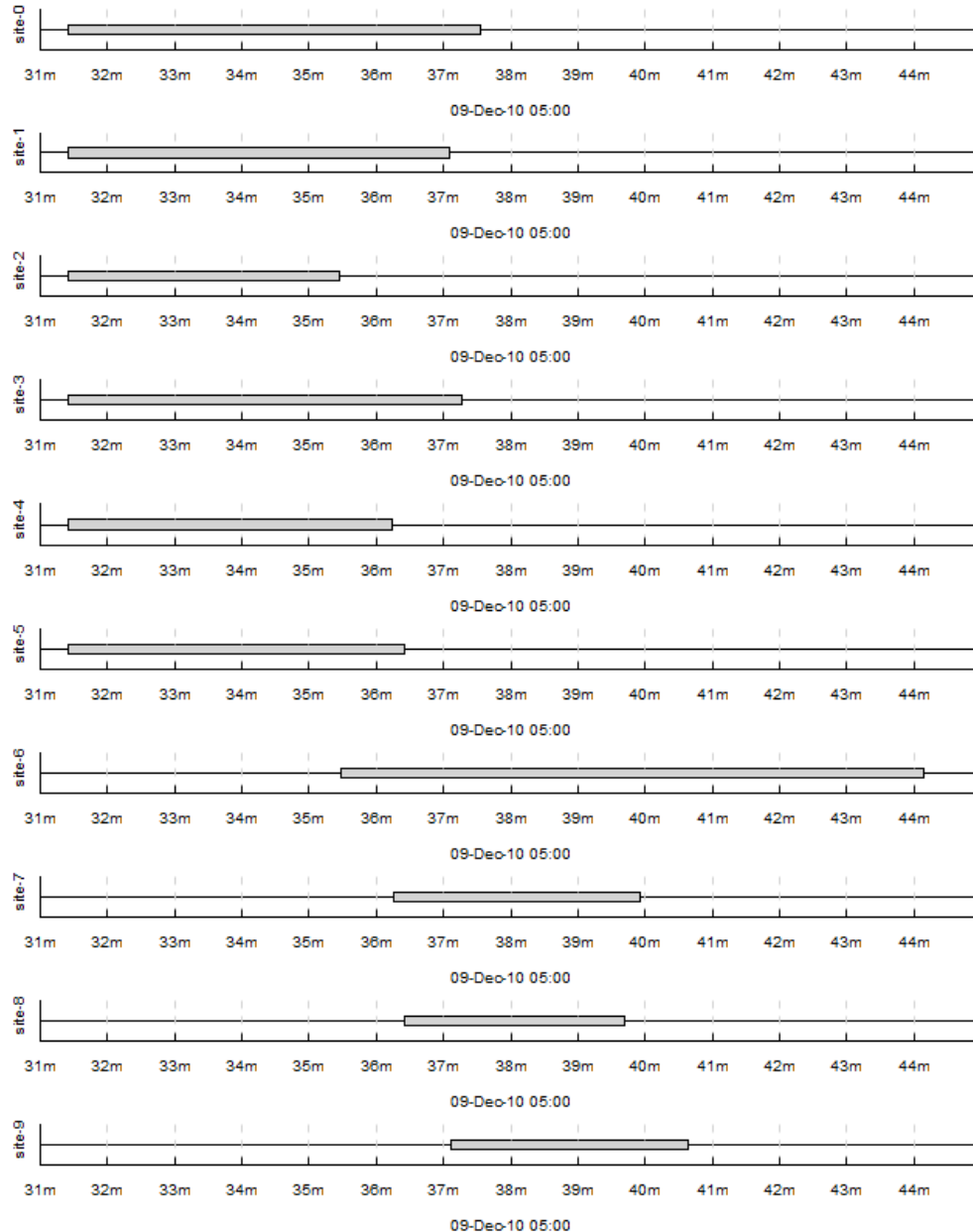
Average count / activity count — `acount N` — slice the time axis into N-second bins; draw histograms of the number of activities specified by `>`, `<` and `!` in each bin. More specifically:

- Each input track mapping to this output track is interpreted as a counter

- `>` means "+1", `<` means "-1", `!` are counted separately

- In every bin, draw stacked bars: 1 bar per input track (with a consistent coloring across bins), height of the bar is average value of the counter + number of `!` events in this bin (usually you *either* use `>` and `<`, *or* `!`).

Example with 1 input track per output track: `acount 5`
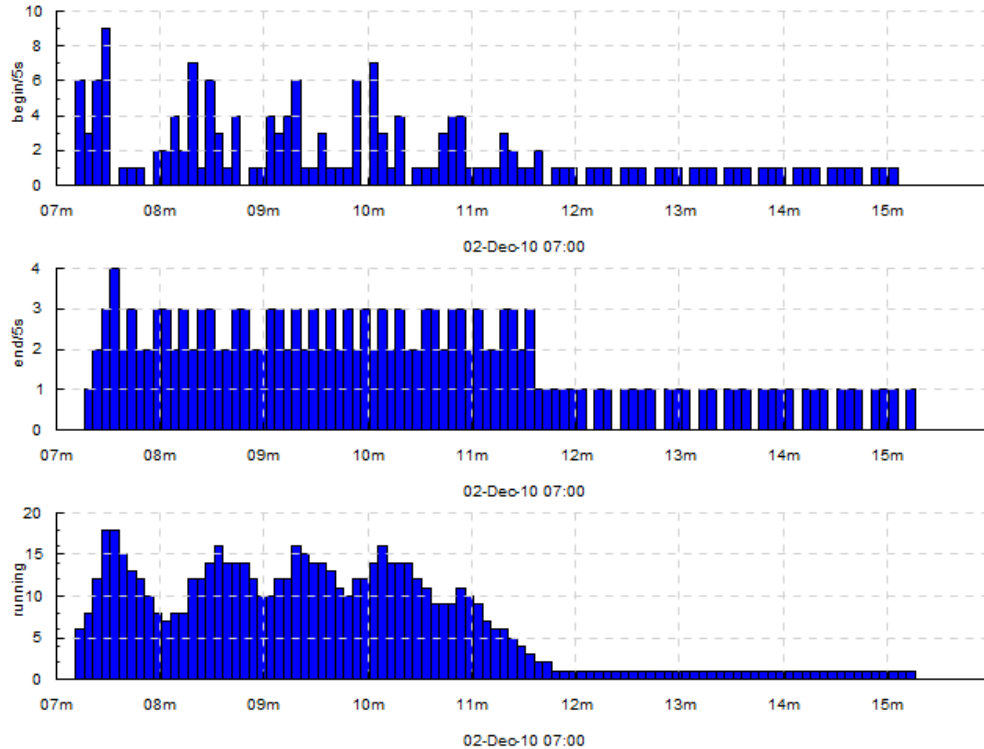
Here, we draw how many tasks are being 1) started 2) finished 3) being currently executed per every 5 seconds. The trace looked like this:

```
...
2010-12-02 07:08:15 !begin/5s
2010-12-02 07:08:15 >running
...
2010-12-02 07:08:18 !end/5s
2010-12-02 07:08:18 <running
...
```

So, for the top two graphs we're counting `!` events and for the bottom graph we're looking at the average value of the counter bumped by `>` and `<`.

Example with two input tracks per output track: `within[.]  acount 5`
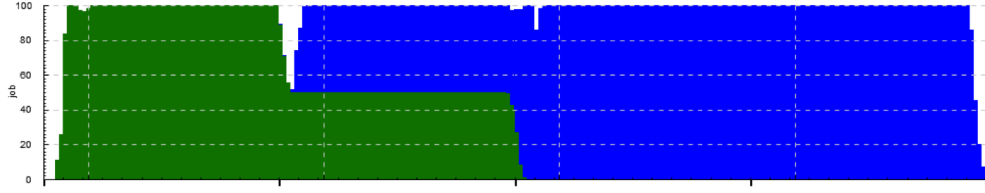
Here we see how the cluster is executing one job at full capacity, then another job comes in, preempts some of the first job's tasks and starts its own tasks there, etc. Each job's task starts are mapped to `>job.JOBID` and task completions are mapped to `<job.JOBID`.

Example with several input tracks per output track: `within[.]   acount 5`
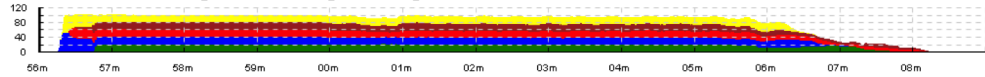
Here, a single job is being executed on a cluster with a task queue sharded into 4 sub-queues. The graph shows the number of concurrently executing tasks taken from each shard over time (every worker is attached to a single shard; when a task starts on a worker attached to queue shard S, an event `>run.S` is emitted; when it finishes, `<run.S` is emitted). We see that some shards deplete later than others.

Average activity count as percentage — `apercent N X` — exactly the same as `acount N`, but the Y axis is scaled into percentages of X. Useful if e.g. you have X cores and you're counting what percent of them are busy running tasks delimited by `>` and `<` at any given moment.

Example with both 1 and several input tracks per output track: `+k run 'apercent 60 420' +k run 'within[.]  apercent 60 420'`
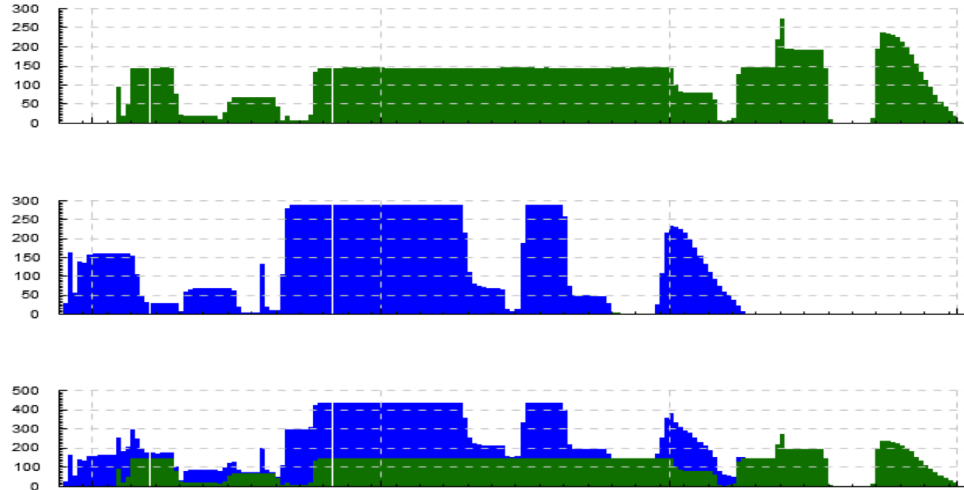

The original log has been lost and only a picture with manually erased axis labels was left. Originally, there were labels on all axes. 1 tick on the X axis is actually 1 hour.

Here, two jobs are being executed on a cluster with 420 cores. The graphs show utilization of the cluster (what percentage of cores are busy at any given moment) by both jobs separately and together. When a task starts, an event `>run.JOBID` is emitted; when a task completes, `<run.JOBID` is emitted. To understand how we get three plots here (1 per job and 2 for both together), see section Track and plot kind mapping.

Average relative activity frequency — `afreq N` — same as `acount N`, but in each bin the heights of bars for different input tracks are normalized to add up to 1. So, this graph is useless if you have just 1 input track mapped to the output track, but if you have many, it shows you the ratio between intensity of different activities.

Example with two input tracks per output track: `within[.]  acount 5`
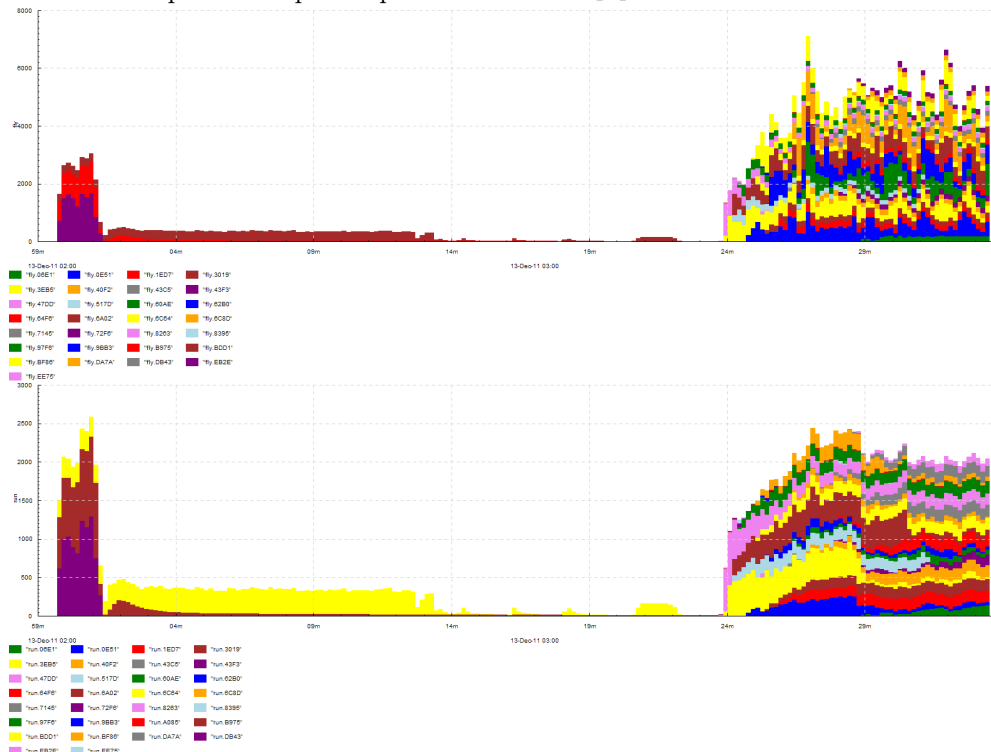

Here, two jobs are executing on the cluster, each job's task starts are mapped to `>absolute.JOBID` and `>relative.JOBID` and task completions are mapped to `<absolute.JOBID` and `<relative.JOBID`. Then `relative` is drawn with `within[.]  afreq 5` and `absolute` with `within[.]  acount 5: -k relative 'within[.]  afreq 5' -k absolute 'within[.]  acount 5'`[2]. Both graphs illuminate different interesting aspects of the data.

---

[2]In the latest versions of **timeplot** you don't need this data duplication, you can use `+k` — see section Track and plot kind mapping

2.5.4 Kinds for discrete data

Discrete value frequency — `freq N TYPE` — slice the time axis into N-second bind; within each bin, draw bars proportionally to the frequencies of different values of a discrete variable (` events). Multiple input tracks per output track are not supported. If TYPE is `clustered`, the bars are drawn next to each other; if TYPE is `stacked` (default), they are stacked vertically and add up to 1. **Discrete value histogram** — `hist N TYPE` — same as `freq N TYPE`, but absolute numbers of occuresnces of each value are drawn rather than the frequencies.

Example: `hist 60`
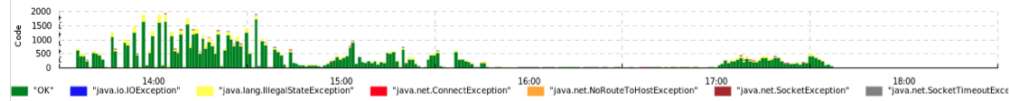
Here, we draw the frequency of different outcomes of page downloads by a web crawler written in Java. The input trace consists of events like `=Code 'OK`, `=Code 'java.io.IOException` etc.

These two kinds can be emulated by `acount` and `afreq`: a chart of kind `freq N` over events `=foo 'BAR` is equivalent to a chart of kind `within[.]  afreq N` over events `!foo.BAR`.

2.6 Track and plot kind mapping

The process of mapping input events to output tracks consists of several steps. We'll understand it by describing the mapping algorithm and demonstrating it on a few simple examples involving the different parts of the algorithm.

Here's what happens to every input event (assume the event's input track is TRACK):

- Compare it to all the patterns among `+k PATTERN '+SUF TYPE'`. For those that match:

    If TYPE is `within[#] SUBTYPE`, and TRACK is of the form `BASE#SUB`, place the event onto output track `BASE.SUF`.

    Otherwise, place the event onto output track `TRACK.SUF`.

- Do the same for `+dk '+SUF TYPE'`.

- Do the same for the *first* matching pattern among `-k PATTERN '+SUF TYPE'`.

- If no `-k` patterns matched, do the same for `-dk`.

To understand this, compare it to the simple special cases shown below.

Simplest case, single input track and single plot type: 1 input track, 1 plot kind mapping — the "default" mapping specified by `-dk K`. This sole input track should be drawn with plot kind K (e.g. `dots`). We'll have 1 dot plot in the output, showing events from the sole input track.

Multiple input tracks, single plot type: Several input tracks, 1 default plot kind mapping. We get several plots of the same type (e.g. several dot plots) vertically stacked with a common time axis. **For example:** if we have several memcached servers and we measure the durations of requests to them, we can have input tracks named `rtime.mcd1`, `rtime.mcd2` etc.

Multiple input tracks with different plot types: We wish to draw some input tracks with one plot type and some with another. We specify several regex/type mappings using `-k PATTERN TYPE`.

E.g. if we also have an input track `cache` with discrete measurement events `'HIT` and `'MISS` and we wish to draw their rate, we can use `-k rtime dots -k cache 'freq 1'`. Then the output will have vertically stacked dot plots for tracks `rtime.mcd1`, `rtime.mcd2` etc, and a frequency plot for the track `cache`.

Input tracks that don't match any of the `-k` patterns get drawn using the default plot type specified by `-dk`.

Single input track drawn with several different plot types: We wish to make events from a single input track participate in multiple output plots, e.g. draw both the absolute and relative frequency of a track of discrete measurement events. Then we use `+k PATTERN '+SUF TYPE'`. This means "for tracks that match `PATTERN`, append `.SUF` to their name and draw a plot of type `TYPE`".

For example: `+k cache '+f freq 1' +k cache '+h hist 1'` will produce two output tracks: `cache.f` drawn with plot type `freq 1` and `cache.h` drawn with plot type `hist 1`. Or, if (for some reason) you decide to draw both a dot and line plot of request times, you can try `+k rtime '+dot dots' +k rtime '+line lines'` and get tracks `rtime.mcd1.dot`, `rtime.mcd2.dot` ... drawn with `dots` and `rtime.mcd1.line`... drawn with `lines`.

There's also `+dk '+SUF TYPE'` with similar semantics.

The suffixes are needed because otherwise the names of output tracks for the different plot types specified by `+k` for a single input track would be identical, i.e. it would be the same output track, **timeplot** cannot have two identically named output tracks. If the suffix is not specified (e.g. `+k rtime dots`) it is assumed to be empty.

Events from several similarly named input tracks drawn on a single plot: Assume that you wish to draw dot plots of request times to different memcached servers not on several vertically stacked plots, but on a single dot plot, with different servers color-coded. Assume again that the input tracks are named `rtime.SERVER`.

Then you should use the `within[SEP]` plot meta-type: `-k rtime 'within[.] dots'`. This means: map input tracks to output tracks by dropping everything after the separator `SEP`, in this case after `.`, and we get a single output track `rtime` to which go all the events from `rtime.mcd1`, `rtime.mcd2` etc. The plot type `dots` will then take care of color-coding the different input tracks within a single output track. Other plot types deal with this situation in a sensible manner too, see section Plot kinds.

2.7 Option reference

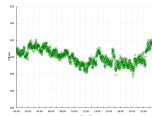| Option	Meaning	Default value
`--help`	Show help	
`--version`	Show version information	
`-if INFILE`	Input filename. "`-`" means read from stdin. Do not use "`-`" for large inputs (above ≈100,000 events)! **timeplot** can only work well on large inputs if it is in a file.	*Required*
`-o OUTFILE`	Output filename with extension	*Required*
`-of FORMAT`	Output format: `svg`, `png`, `pdf`, `ps`	Extension of `-o`
`-or WIDTHxHEIGHT`	Output resolution, e.g. `640x480`	Extension of `-o`
`-tf PATTERN`	Format of time in the input file as in man strptime but with fractional seconds supported via `\%OS` - will parse `12.4039` or `12,4039`. Also, `%^[+-][N]s` will parse seconds since the epoch, for example `%^-3s` are milliseconds since the epoch (N can only be 1 digit)	`%Y-%m-%d %H:%M:%OS`
`-dk KIND`	Default diagram kind. Do not forget to quote it! See section Track and plot kind mapping.	None
`+dk KIND`	—	None
`-k PATTERN KIND`	Diagram kind for pattern PATTERN. See section Track and plot kind mapping.	None
`+k PATTERN KIND`	—	None
`-fromTime TIME`	Filter events whose time is ≥ this time. Format specified by `-tf`.	None (no filter)
`-toTime TIME`	Filter events whose time is ≤ this time. Format specified by `-tf`.	None (no filter)
`-baseTime TIME`	Draw time axis labels as seconds elapsed since TIME, instead of absolute time. Format specified by `-tf`.	*Required*

2.8 Example data and exercises

In this section we list some example datasources and exercises to try on them.

2.8.1 Consumer power consumption.

http://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption. This dataset contains measurements of power consumption of a single household, measuring several characteristics over the course of several years. It's in a simple CSV format which is described at the link. Beware: some data is not available and there are question marks in place of missing data items. You can completely skip such rows for the purpose of this exercise.

Excerpt from the log:

```
Date;Time;Global_active_power;Global_reactive_power;Voltage;Global_intensity;Sub_metering_1;Sub_metering
16/12/2006;17:24:00;4.216;0.418;234.840;18.400;0.000;1.000;17.000
16/12/2006;17:25:00;5.360;0.436;233.630;23.000;0.000;1.000;16.000
16/12/2006;17:26:00;5.374;0.498;233.290;23.000;0.000;2.000;17.000
16/12/2006;17:27:00;5.388;0.502;233.740;23.000;0.000;1.000;17.000
16/12/2006;17:28:00;3.666;0.528;235.680;15.800;0.000;1.000;17.000
```

Exercise	*Expected result*
Make a dot plot of `Voltage` for January 1st, 2007. Do not limit *the trace*, use `-fromTime` and `-toTime`.	
Make a line plot of `Global_active_power` vs `Global_reactive_power` on one plot and a line plot of `Voltage` on another (i.e., the result should be a stack of two plots), for January 1st, 2007. Use an opacity level of 0.5.	
Make a quantile plot, with quantiles of 0.1, 0.5 and 0.9 over bins of 1 day, for `Global_active_power` and `Global_reactive_power`, for the entire dataset.	

2.8.2 Video encoding.

This dataset generously provided by Dmitry Popov. TODOupload. This is a set of logs of a video encoding pipeline. For every video sample, the log contains times that this sample entered several steps of the pipeline: `Grab1`, `Grab2`, `SR.YUV` and `AviWr`. The three logs show how the same video was encoded with 3 versions of the program after different optimizations.

Excerpt from the log: This excerpt shows all entries in the log `1.txt` relating to sample `[0]`. In the full log, entries for all samples are intermixed. Use column 1 as the timestamp.

```
0.000000 (0.000000) (0.000000) Grab1 sample [0] 0.000000, locking..
0.000006 (0.000006) (0.000006) Grab1 [0] got lock, carry on..
0.000611 (0.000604) (0.000611) Grab1 [0] sample processed.
0.000615 (0.000000) (0.000615) Par1.In received sample [0]
0.000629 (0.000000) (0.000629) SR.YUV got sample [0]
```
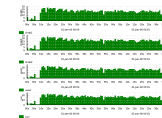
```
0.011636 (0.011007) (0.011636) SR.YUV done with sample [0]
0.011643 (0.000000) (0.011643) ParLast.In received sample [0]
0.011661 (0.000000) (0.011661) Grab2 sample [0] 0.000000, locking..
0.011666 (0.000005) (0.011666) Grab2 [0] got lock, carry on..
0.014947 (0.003280) (0.014947) Grab2 [0] sample processed.
0.598770 (0.000000) (0.598770) AviWr.V video sample [0] received
9.736371 (0.000466) (9.736371) AviWr.V [0] WriteVideo, locking..
9.736373 (0.000001) (9.736373) AviWr.V [0] got lock, writing..
9.736460 (0.000086) (9.736460) AviWr.V [0] write complete.
```
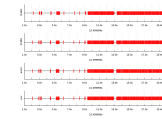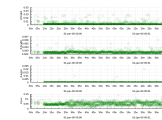
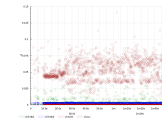| *Exercise*	*Expected result*
Plot the number of samples arriving per second to `Grab1`, `Grab2`, `SR.YUV` and `AviWr`.	
Make an event plot, using impulse events for sample arrivals to these stages. Limit to the first 20 seconds.	
Make 4 dot plots for durations of processing at these stages.	
Plot the same, but on a single plot (hint: use `within`).	

2.8.3 DrWeb log

. TODOprovide. *This log was generously provided by Paul Graphov.* This is a log of DrWeb Antivirus Server, and the most interesting entries in it are those relating to TCP data flow and to DB transactions:

```
20121024.114904.78 db3 [  415    448] mth:1  ...:ANTON-WIN7RU: data arrived, 32b
20121024.114904.78 tr2 [  415    448] mth:1  ...:ANTON-WIN7RU: rcv 17 GETTIME(6348666174470429760) 0b dat
20121024.114904.78 db3 [  415    448] mth:1  ...:ANTON-WIN7RU: cancel timeout, GETTIME
20121024.114904.78 tr2 [  415    448] mth:1  ...:ANTON-WIN7RU: cmd (49b) "11 TIME 6348666174470429760 634
20121024.114904.78 db3 [  415    448] mth:1  ...:ANTON-WIN7RU: cancel timeout, prolongate
20121024.114904.78 db3 [  415    448] mth:1  ...:ANTON-WIN7RU: set timeout to 60000 ms <6348666180478295
20121024.114904.78 db3 [  415    449] mth:2  ...:ANTON-WIN7RU: data arrived, 14b
```

    and

```
20121024.114904.78 db3 [  415    449] mth:2  [DB] Successful BEGIN transaction, 00.000 wait
20121024.114904.78 db3 [  415    449] mth:2  [IntDB] Statement "..."
20121024.114904.78 db3 [  415    449] mth:2  [DB] OK, 00.000, ...
20121024.114904.78 db3 [  415    449] mth:2  [IntDB] Statement "..."
20121024.114904.78 db2 [  415    449] mth:2  [DB] 1 row, 00.000, ...
20121024.114904.78 db3 [  415    449] mth:2  [IntDB] Statement "..."
```

```
20121024.114904.78 db2 [  415    449] mth:2  [DB] 79 rows, 00.000, ...
20121024.114904.78 db3 [  415    449] mth:2  [IntDB] Statement "COMMIT"
20121024.114904.78 db3 [  415    449] mth:2  [DB] Database has been freed but nobody wants it now
20121024.114904.78 db3 [  415    449] mth:2  [DB] Successful COMMIT transaction, 3 statements, 00.000 wai
```
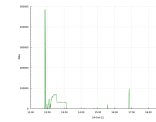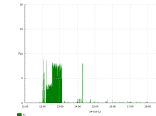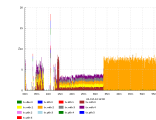
| Exercise	Expected result
Plot the data arrival rate (according to `data arrived` entries) per 5-second bins (use `sum`).	
Plot the transaction commit rate (according to `Successful COMMIT` entries) per 5-second bins.	
Plot the same transaction commit rate, but so that we can see how it adds up from contributions by different threads (`mth:1`, `pth:2` etc.). Limit to the interval from 12:00 to 13:00.	
Same as above, *and use the same trace file*, but make it 1 plot per thread type (`mth`, `pth`, `dbv` etc., but still with a breakdown *by thread* within each type's plot).	
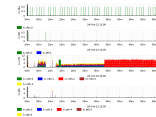
2.9 Gallery

3   splot — visualizing behavior of many concurrent processes

splot draws a single Gantt-like chart with a birds-eye view of the activity of a number of concurrent processes, each of which may be in one of several different states at each moment (e.g. processing one of several jobs, or being in a particular stage of processing).

The input to **splot**, in the basic form, is a sequence of events "activity started", "activity finished" (specifying the timestamp and the process whose activity we're speaking about). Every event also includes a color with which to render the activity.

3.1   Simple example

This example has two tracks "thread-1" and "thread-2" and demonstrates most of the basic features.

```
2010-10-21 16:45:12.014 >thread-2 blue
2010-10-21 16:45:13.329 >thread-1 blue
2010-10-21 16:45:13.635 <thread-1
2010-10-21 16:45:13.800 <thread-2
2010-10-21 16:45:13.810 >thread-1 blue
2010-10-21 16:45:13.810 >thread-2 orange
2010-10-21 16:45:14.010 >thread-1 orange
2010-10-21 16:45:14.258 <thread-2
2010-10-21 16:45:14.623 <thread-1
2010-10-21 16:45:14.629 >thread-2 orange
2010-10-21 16:45:15.138 >thread-1 orange
2010-10-21 16:45:15.319 >thread-1 blue
2010-10-21 16:45:15.502 <thread-1 red
2010-10-21 16:45:15.502 !thread-1 black Hi there
2010-10-21 16:45:16.112 <thread-2
```

Invoke **splot**:

```
$ splot -if splot.simplest-example.trace \
        -o splot.simplest-example.png -w 400 -h 160
```

Result:

Origin at 2010-10-21 16:45:12.014, 1 small tick = 1000.0ms

35
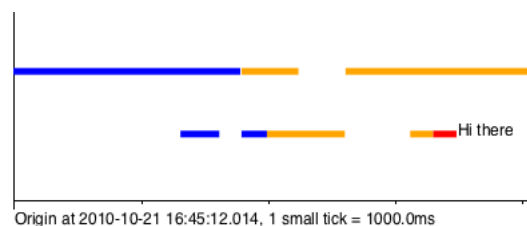
3.2 Motivation

In this section we shall consider the plot we already saw in the introduction and see how it helps highlight a number of problems in the program whose behavior was visualized, that would be difficult to find without **splot**.
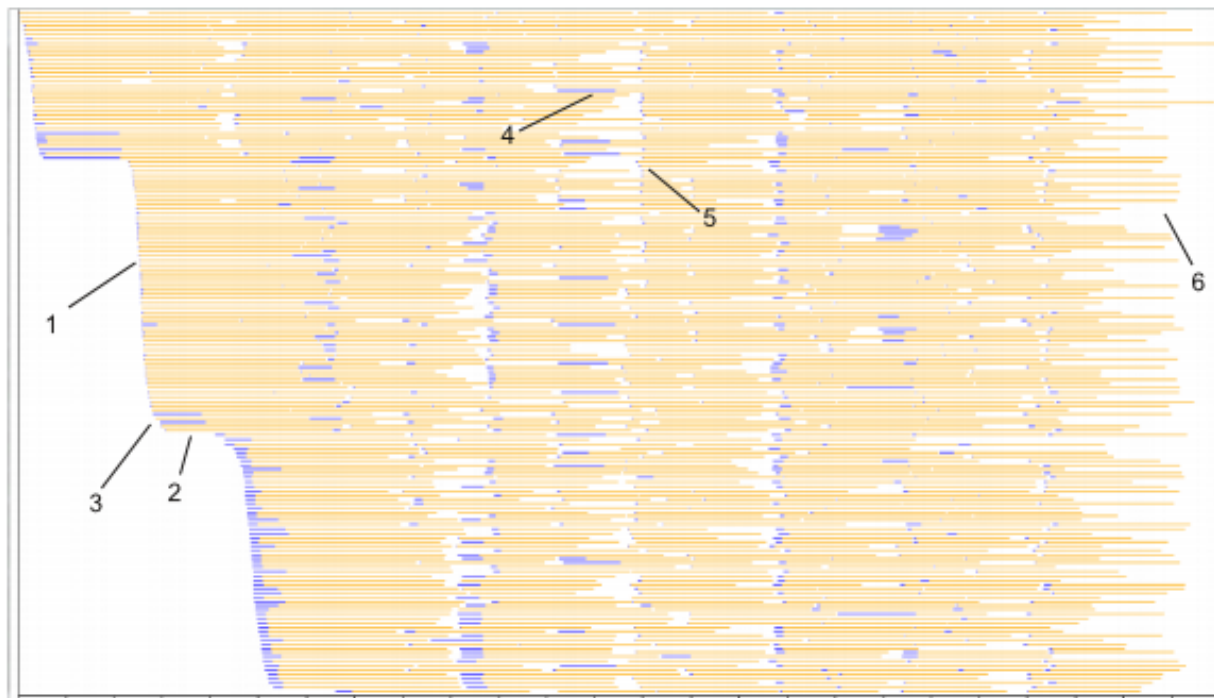
Recall that here, the X axis is time, the Y axis is cluster worker (core), orange is computations and blue is fetching data from memcached.

There are several anomalies on this picture:

1. Whitespace on the left: Slopes. The beginning of the picture corresponds to the program startup — the first tasks are being fed into the shared queue and workers pick them up and start executing. The whitespace means that a large fraction of the program execution time is spent just warming up, while the workers essentially do nothing. The slopes mean that workers do not receive the first tasks instantly. Either it takes time to pick them from the queue, or they are being fed to the queue not quickly enough.

2. Whitespace on the left: Plateaus. The plateaus mean that there are moments when no tasks are being picked up at all. Either the queue is hung up, or the task producer. We also see how big an impact this has on the overall cluster utilization: a single 1-second plateau is worth 160 seconds of computations (as there are 160 workers on this picture)!

3. The slope on the left is nonlinear. The slope becomes less vertical at the end of each slope line. This means that task pickup rate (or perhaps task generation rate) is not constant. There's a "tail" in task delivery times.

4. Some blue bars are rather long. This means that memcached fetches sometimes take a long time — comparable with computation time. We should optimize them.

5. Vertical patterns of white space and blue bars lining up on their left edge. This means that there are moments when everyone's got nothing to do and the task queue is empty, and then suddenly

a lot of tasks appear in the queue and everyone is busy again. This is probably a problem with the task producer — maybe it is sending tasks in batches, or something like that.

6. A lot of white space on the right. This means that in the end of the program execution, when all tasks are already in the queue and no new ones will appear, a lot of time is spend when faster workers wait for slower workers. The program finishes when the very last worker finishes. The fraction of this whitespace is very well worth 5-10% of the total program time. This anomaly is called "the long tail effect" and can be eased by increasing the task granularity (i.e. submitting many short tasks rather than several long ones). However, this will obviously increase load on the queue, so we have a trade-off here.

We see that a simple picture of the cluster behavior showed us quite a few non-trivial things, most of which would be very difficult to find without a visualization.

3.3   Concepts

Let us now consider in detail all the concepts necessary to understand and use **splot**.

Track (also process) An entity which changes its state between several values (performs several activities) over time. At any moment, a process is doing at most one activity. We're usually interested in seeing the visual pattern emerging between different tracks. For example, it is often meaningful to assign 1 track = 1 thread in a multi-threaded program (in a distributed program one should of course include the machine into the track id).

Activity A period during which a process is in a particular problem-specific logical state. A single activity is drawn with a single color, as a bar which is horizontally as long as the activity (if there are a lot of tracks, this bar can be as thin as a hairline). For example, if we have a program whose threads are either computing, doing IO or waiting idly, we may depict the 'computing' activity with orange color, IO as blue and idle waiting as lack of activity (no color at all).

Event Mark of the beginning or end of an activity on a particular track, specifying the activity's color. There are also "text" events which allow to draw text markers above the usual colored bars. An event has a timestamp, track and event type (activity start / activity end / text). For example, when our hypothetical program starts doing IO in thread T, we should represent this in **splot**'s input as an event "start blue activity on track ".

Color Colors in **splot** can be specified in several ways. In the simplest form, it can be an SVG color name or hex code (e.g. "red" or "lightblue" or "#ff0033"), it can be an arbitrary string (then a random color will be generated so that different strings correspond to different colors — this is useful to color-code an unknown number of different types of activities, e.g. if you have worker processes servicing several clients and you wish to get a picture of who services whom and color-code the clients: "client-5"), and it can be an arbitrary string within a color scheme (e.g. "/success/client-5" or "/failure/client-5", and you might define the "success" colorscheme to consist of several greenish colors and "failure" of reddish).

Color scheme A list of colors which will be cycled between when generating colors for tracks whose color is not specified explicitly by a SVG color name or hex code.

3.4 Input format

The input to **splot** consists of a series of events.

Format	Meaning
`TIME >TRK COLOR`	Start activity of color COLOR on track TRK at time TIME (if there already is an activity, finish it and start a new one instead).
`2010-10-21 16:45:09,431 >r2b3.t5 blue`	
`TIME <TRK`	Finish the current activity of track TRK at time TIME
`2010-10-21 16:45:10,322 <r23.t5`	
`TIME <TRK COLOR`	Finish the current activity of track TRK at time TIME, overriding the color it was given at the start by COLOR. This is useful, e.g. do indicate that an activity failed by drawing it with red color (obviously when the activity begins, we don't know if it will fail or not).
`2010-10-21 16:45:10,322 <r2b3.t5 red`	
`TIME !TRK COLOR TXT`	Draw text TXT with color COL on track TRK, left-justified at time TIME
`2010-10-21 16:45:10,322 !r2b3.t5 black read()`	

3.5 Advanced features

3.5.1 Bar height

Bar height is specified with `-bh`: either `-bh fill` or `-bh HEIGHT` (e.g. -bh 5). `fill` means "set bar height to fill the whole vertical space". Consider the previous simple example, here it is with `-bh fill`:
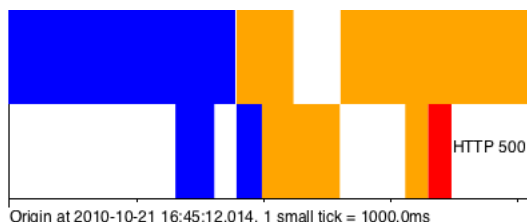

Figure 5: A trivial example of **splot** usage with the `-bh fill` option

`-bh fill` is vital when you have a lot of tracks (at least dozens). Perhaps it should be the default.

3.5.2 Expiring activities — if < is missing

In systems where components can crash (which is, most systems :)), it might happen so that your log catches only the beginning of an activity but not its end, because the component has crashed in the middle. You can tell **splot** "expire all activities if they take longer than X seconds" by using `-expire X`; then, if an activity has not finished within X seconds, **splot** will draw a dashed line and an X marker, meaning that the process probably crashed somewhere on the dashed line[3].

Consider a small example similar to the one we had before.

```
2010-10-21 16:45:12.014 >worker-2 blue
2010-10-21 16:45:13.329 >worker-1 blue
2010-10-21 16:45:13.635 <worker-1
2010-10-21 16:45:13.800 <worker-2
2010-10-21 16:45:13.810 >worker-1 blue
2010-10-21 16:45:13.810 >worker-2 orange
2010-10-21 16:45:14.010 >worker-1 orange
2010-10-21 16:45:14.258 <worker-2
2010-10-21 16:45:14.623 <worker-1
2010-10-21 16:45:14.629 >worker-2 orange
2010-10-21 16:45:15.138 >worker-1 orange
2010-10-21 16:45:15.319 >worker-1 blue
2010-10-21 16:45:16.512 <worker-2
2010-10-21 16:45:18.412 >worker-2 blue
2010-10-21 16:45:20.112 <worker-2
```

Let us draw it with an expiration time of 2000 milliseconds:

```
$ splot -if splot.expire-example.trace -o splot-expire-example.png \
        -expire 2000 -bh 15 -w 400 -h 160
```
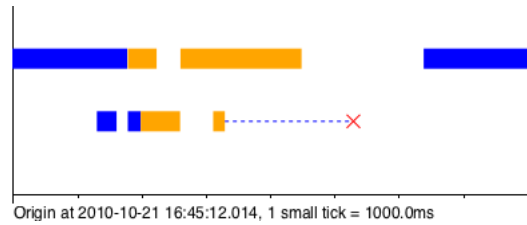
Origin at 2010-10-21 16:45:12.014, 1 small tick = 1000.0ms

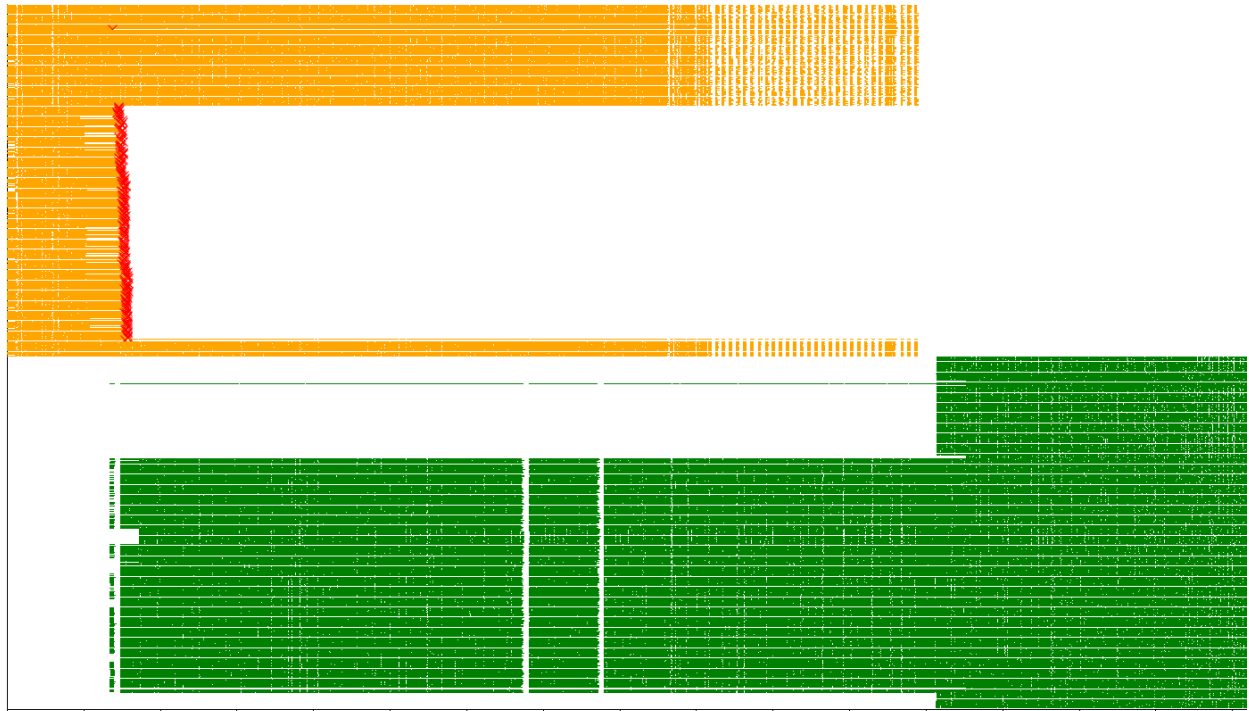Figure 6: Using the `-expire` option with **splot**

Figure 7: Using the `-expire` option with **splot**: a real-life example

Here's what we get:

Here we see that perhaps "worker-1" (here drawn second, as its first event happened later than worker-2's) crashed and that's why it didn't do anything in the later parts of the graph.

Figure 7 shows a more complex real-life example from a cluster (the input log has unfortunately been lost). Here a large number of workers are preempted at once.

3.5.3 Phantom color — if > is missing

Sometimes you process logs which start in the middle of the program's execution, so the log doesn't catch the beginning events of activities that were active at the moment the log was started (however, the log *does* catch their finishing events). **splot** can display such "phantom" activities in a color of your choice, using the

---

[3]It's probably meaningful to specify expiration times for each activity separately, but currently **splot** has just a single global option

41

-phantom COLOR option. Specifically, if a track starts with a < event instead of >, then **splot** will assume that there was a > event with color COLOR in the past on this track.

Consider the same example as above, but let us cut it in the beginning, as if we had a truncated log:

```
2010-10-21 16:45:14.010 >worker-1 orange
2010-10-21 16:45:14.258 <worker-2
2010-10-21 16:45:14.623 <worker-1
2010-10-21 16:45:14.629 >worker-2 orange
2010-10-21 16:45:15.138 >worker-1 orange
2010-10-21 16:45:15.319 >worker-1 blue
2010-10-21 16:45:16.512 <worker-2
2010-10-21 16:45:18.412 >worker-2 blue
2010-10-21 16:45:20.112 <worker-2
```

Now invoke **splot**:

```
$ splot -if splot-phantom-simple-example.trace \
        -o splot-phantom-simple-example.png \
        -phantom gray -w 400 -h 160 -bh 5
```
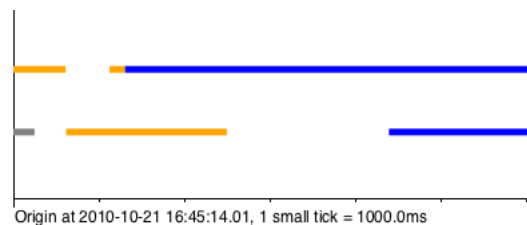
And here's what we get:

Origin at 2010-10-21 16:45:14.01, 1 small tick = 1000.0ms

Figure 8: Using the -phantom option with **splot**

We see a gray bar in the beginning of worker-2's track. This is because the first event on this track was `2010-10-21 16:45:14.258 <worker-2`, i.e. an activity closing event, which means that the opening event was missed.

3.5.4   Color auto-generation

How do we color-code activities in **splot**'s input if the set of possible activity types is not known in advance? E.g. you have a set a of worker processes which service a fairly small but unknown number of clients. You assign tracks to worker processes, and you wish to color-code the clients to see a picture of who services whom and when.
In this case, you can simply use the client's id as color: for colors that do not parse as SVG color names or hex codes, **splot** will generate a random color from a default contrast color scheme.
Let us consider a real-life example. In this example, again some workers are processing tasks, but from different clients. The trace looks like this:

```
...
2011-07-27 02:42:03.485 <RACK5UNIT067.37dc
2011-07-27 02:42:03.492 >RACK2UNIT067.4ca8 8610
```

```
2011-07-27 02:42:03.495 >RACK4UNIT075.5b15 8610
2011-07-27 02:42:03.496 >RACK4UNIT067.ec72 877B
2011-07-27 02:42:03.496 >RACK4UNIT067.f3fe 877B
2011-07-27 02:42:03.496 >RACK5UNIT017.0c21 8610
2011-07-27 02:42:03.498 >RACK2UNIT030.9e7a 071C
...
```

So, we use track names of the form `MACHINE.WORKERID` and instead of using color at `>` events, we use the
client ID, asking **splot** to color-code clients for us. The result is shown on figure 9. Here red color denotes
tasks that completed unsuccessfully and other colors (e.g. green and blue; red is excluded from the default
color scheme precisely for situations like this) encode different clients[4].

Origin at 2011-07-27 02:42:00.445, 1 small tick = 60000.0ms
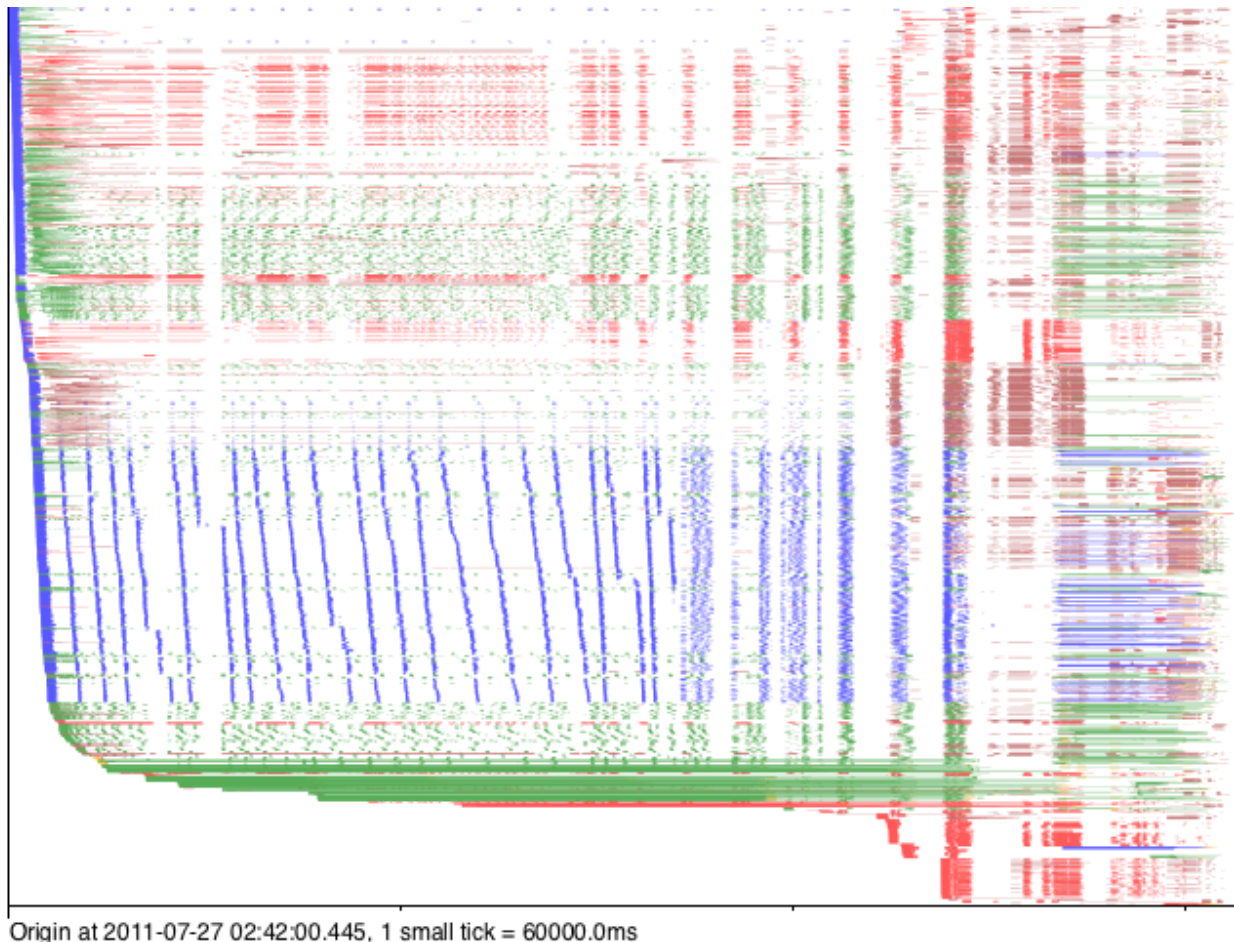
Figure 9: Automatic color generation with **splot**

Assume the same case as above with worker processes servicing different clients. Assume also that worker
processes might be in two modes: regular and workstealing (if they have nothing to do with their current
client, they try to service tasks of some other client). We wish to depict regular tasks and tasks picked by
work-stealing in different shades: e.g. regular tasks as bright colors and work-stolen ones as pale. We still wish
to use color generation for both. This can be achieved by using color schemes. Specifically, we'll color-core reg-
ular tasks by `TIMESTAMP >WORKER CLIENT` and work-stolen tasks by `TIMESTAMP <WORKER /ws/CLIENT`. This

---

[4]Yes, as this rather crazy picture might hint, the program *did* have lots of problems — exposing them, that's what **splot** is
for.

tells **splot** to generate colors for regular tasks from the default color scheme and for work-stolen tasks from the `ws` color scheme, which we must specify in `-colorscheme` parameters, e.g. `-colorscheme ws='lightblue lightgray lightg` (use more colors if you wish to distinguish between more clients).

(unfortunately, an example log illustrating this has been lost)

3.6 Option reference

Option	Meaning	Default value
`-if INFILE`	Input filename	*Required*
`-o PNGFILE`	Output filename	*Required*
`-w WIDTH`	Output width, pixels	640
`-h HEIGHT`	Output height, pixels	480
`-bh BARHEIGHT`	Vertical height of each track's activity bars, pixels, or `fill` to use all the available vertical space	`fill`
`-tf PATTERN`	Format of time in the input file as in man strptime but with fractional seconds supported via `\%OS` - will parse `12.4039` or `12,4039`. Also, `%^[+-][N]s` will parse seconds since the epoch, for example `%^-3s` are milliseconds since the epoch (N can only be 1 digit)	`%Y-%m-%d %H:%M:%OS`
`-tickInterval MILLIS`	Ticks on the X axis will be this often	1000
`-largeTickFreq N`	Every N'th tick will be larger than the others	10
`-sort SORT`	Sort tracks by time of first event (`-sort time`) or by track name (`-sort name`) — see "track sorting" above	`name`
`-expire MILLIS`	Expire activities that do not finish within MILLIS milliseconds — see "expiring activities" above	*none (don't expire)*
`-phantom COLOR`	Set the phantom color which is used if the first event on a track is `<` — see "phantom color" above	*none (no phantom color)*
`-fromTime TIME`	Clip the picture on the left (time in the format of `-tf`, i.e. same as in the input)	*none (don't clip)*
`-toTime TIME`	Clip the picture on the right (time in the format of `-tf`, i.e. same as in the input)	*none (don't clip)*
`-numTracks N`	Explicitly specify the number of tracks for better performance on very large data (see section "Performance" below)	*none (compute from input)*

Option	Meaning	Default value
`-colorscheme SCHEME COLORS`	Declare a colorscheme (see "Color schemes" above). Can be used multiple times. Scheme is an arbitrary string, e.g. `pale` or `bright`. COLORS is a space-separated list of colors in SVG or hex, e.g. `'red green 0x0000FF'`	*none*

3.7   Gallery

Figures 10–20 show a number of pictures produced by **splot** in various real-life situations. Most of them look really creepy and expose different kinds of performance problems in the programs whose behavior they depict. The author already does not remember the precise reasons for the problems — think of it as a horror museum exposition and use your imagination.

Origin at 2011-06-27 05:19:31.683, 1 small tick = 60000.0ms
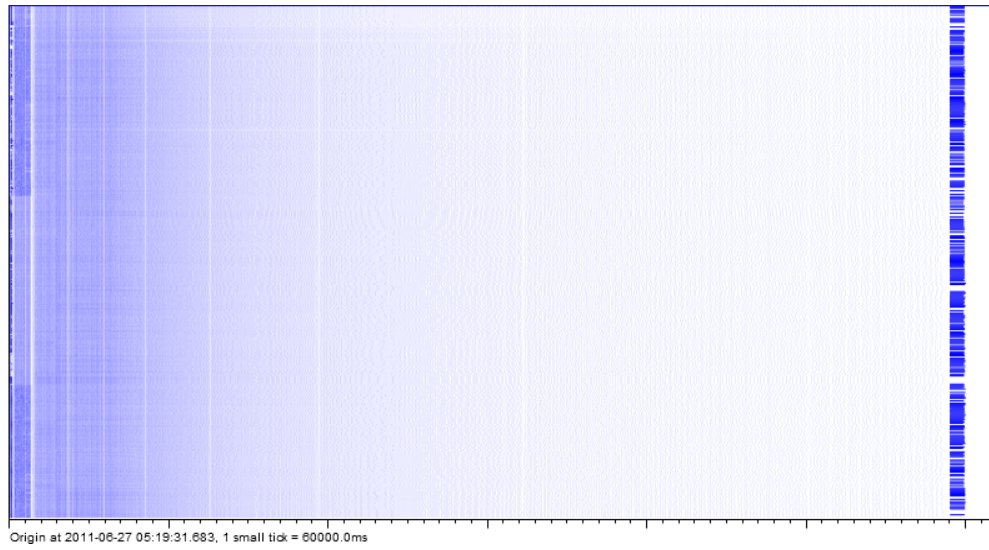
Figure 10: Blue: working, white: waiting. The task queue's performance was gradually becoming worse and worse.

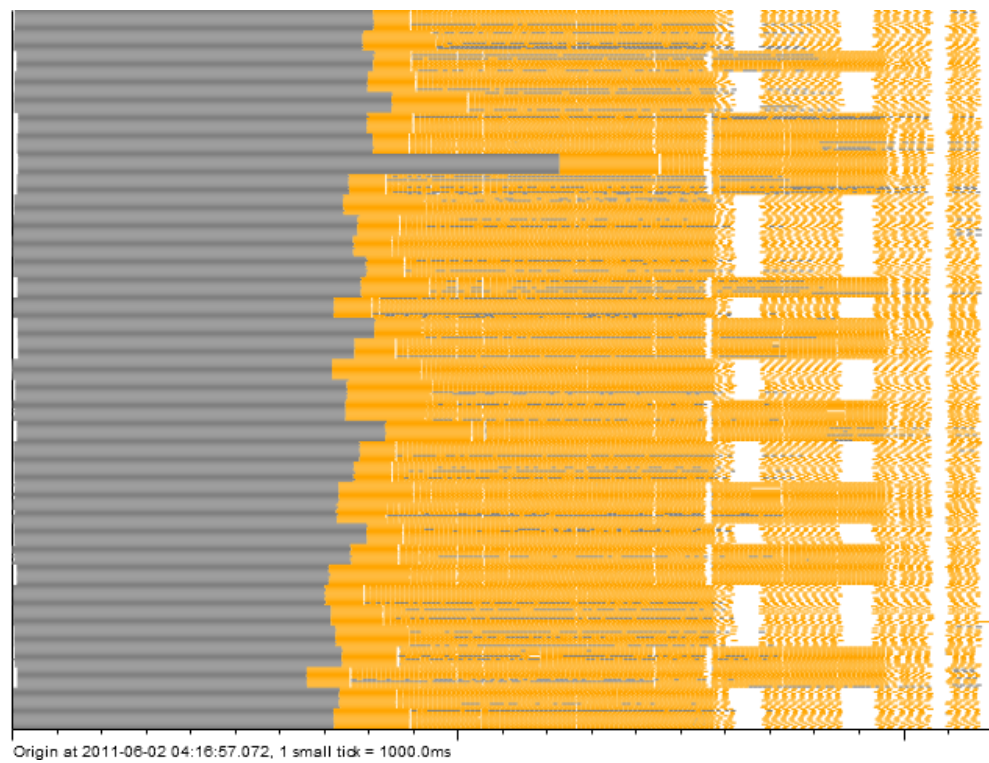Origin at 2011-06-02 04:16:57.072, 1 small tick = 1000.0ms

Figure 11: Orange: working, gray: starting, white: waiting. The first time a program loads, it takes a long time. Loads on the same machine take the same time — .NET DLL caching (here the sort-by-track-name option is used, currently absent until reimplemented).

Figure 12: Colors encode which shard of a task queue was being used. Initially the shards are fast, then they run slower but smoothly. Everything's fine.
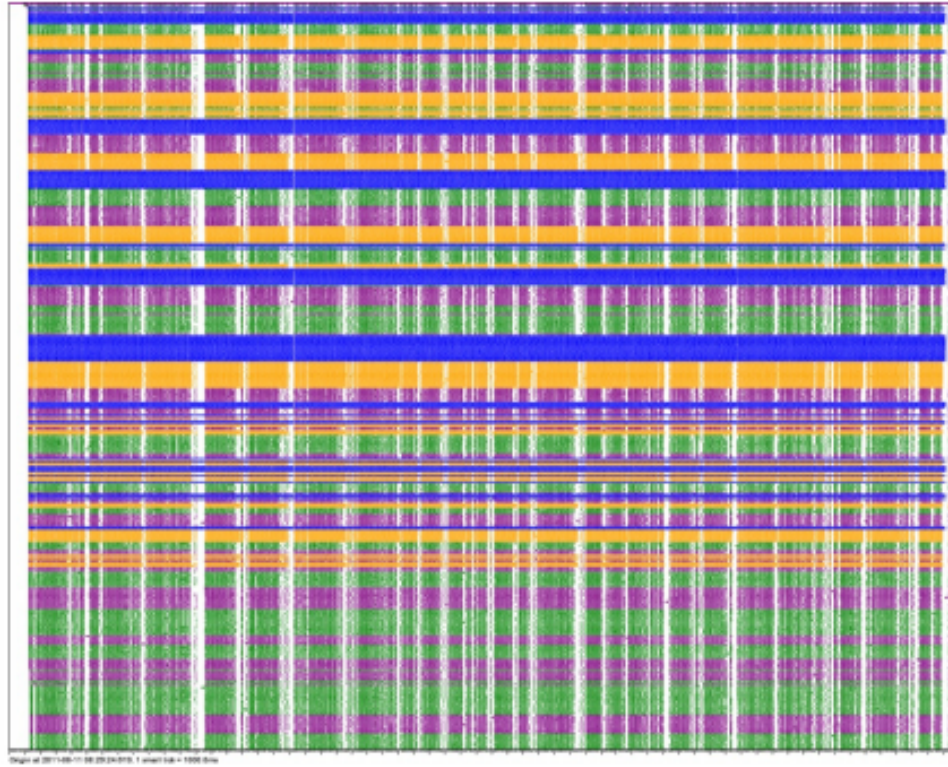
Figure 13: Colors encode which shard of a task queue was being used. Green and purple have problems, yellow and orange don't. Turned out green and purple corresponded to the same physical queue server which had to sustain double load.
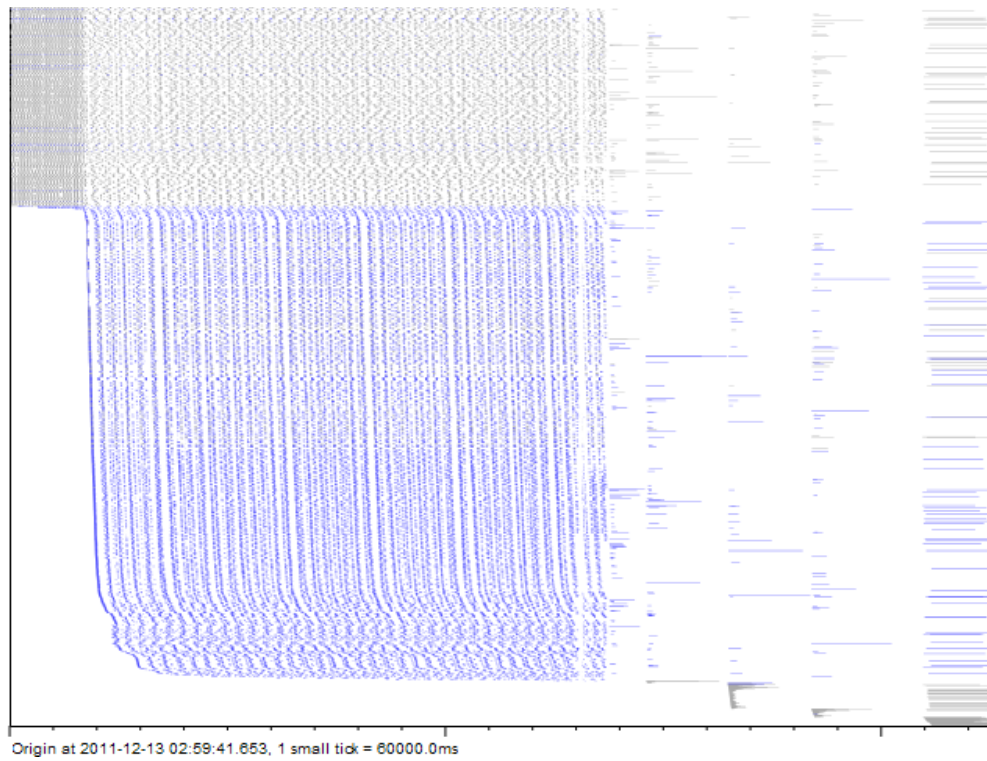
Origin at 2011-12-13 02:59:41.653, 1 small tick = 60000.0ms

Figure 14: Gray: normal tasks, blue: tasks picked by workstealing. After some period, workstealing begins and many workers start processing the job — slowing down the queue, but total throughput is higher.
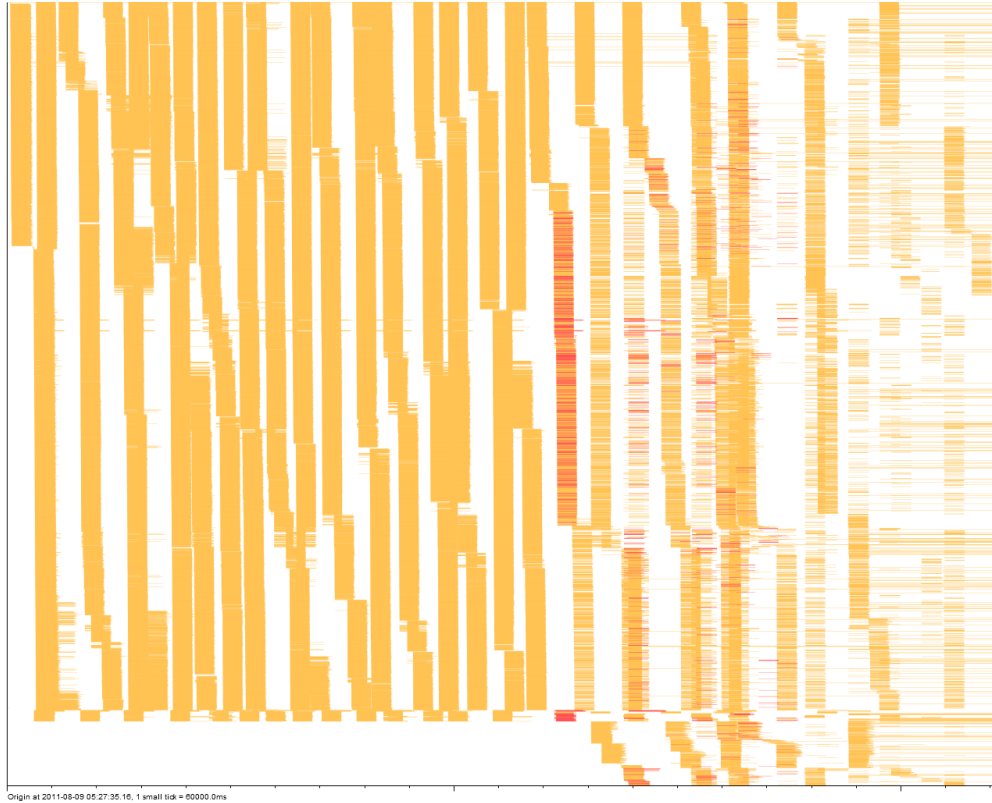
49

Origin at 2011-08-09 05:27:35.16, 1 small tick = 60000.0ms

Figure 15: Orange: working, red: preempted and lost work. The task queue prefetch feature was broken, leading to very strange patterns of task queue utilization.

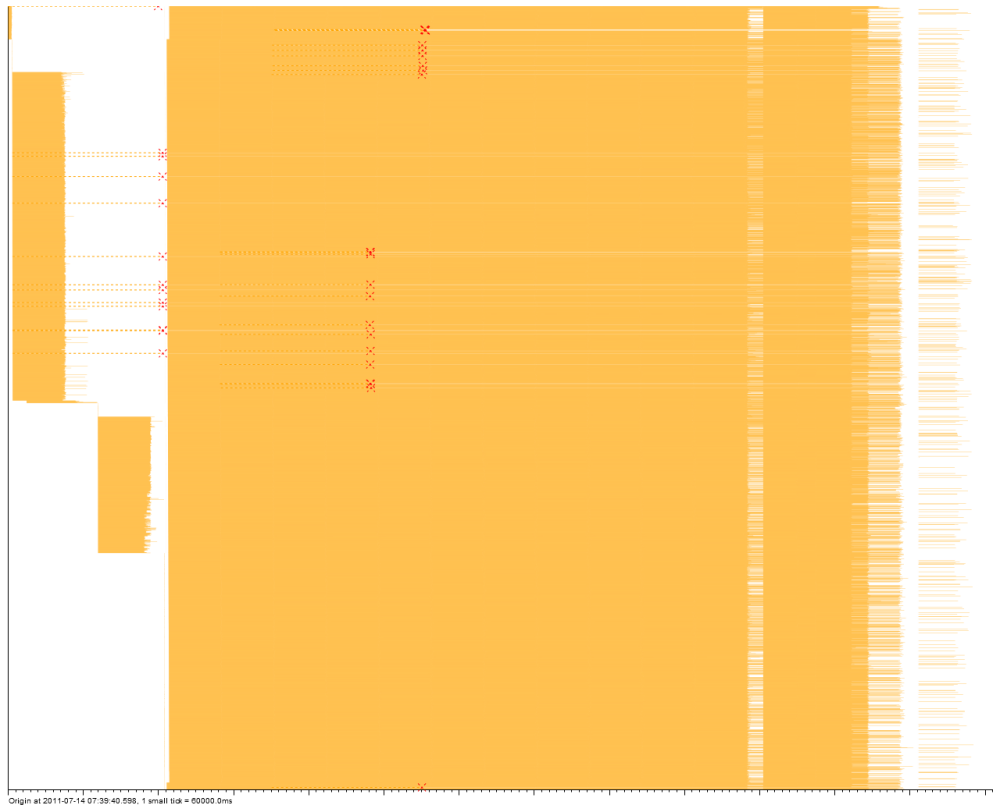Origin at 2011-07-14 07:39:40.598, 1 small tick = 60000.0ms

Figure 16: A simple 2-stage job: a couple of insufficiently parallel data preparation steps, then a long stream of tasks utilizing the cluster well. Two worker machines died in the process.
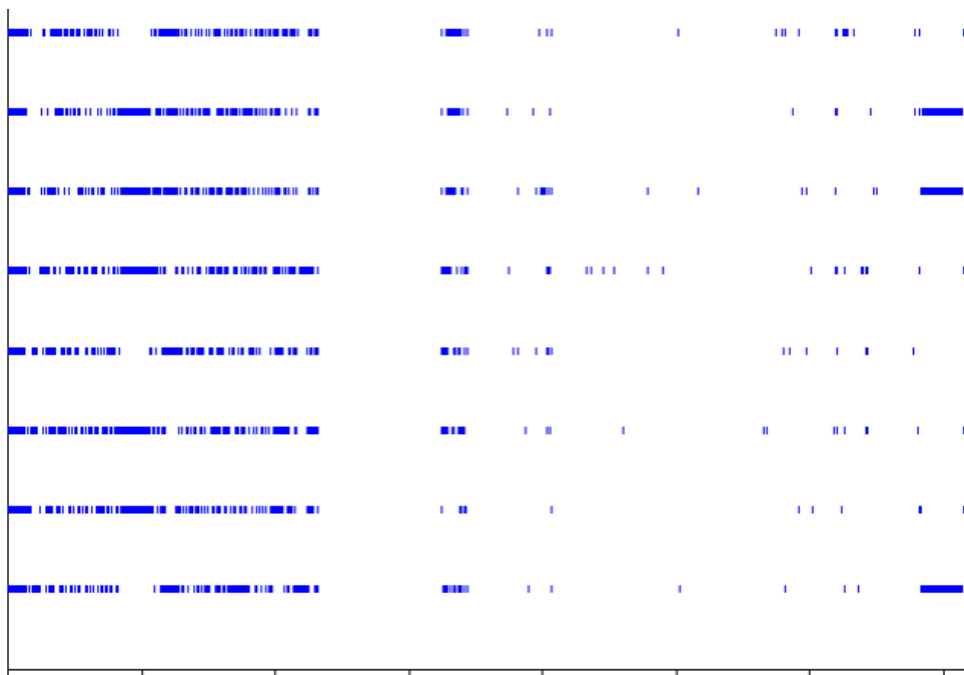
Figure 17: Bars correspond to a web service being called from different threads. Apparently there are periods when it takes longer, and periods when it's not called at all.
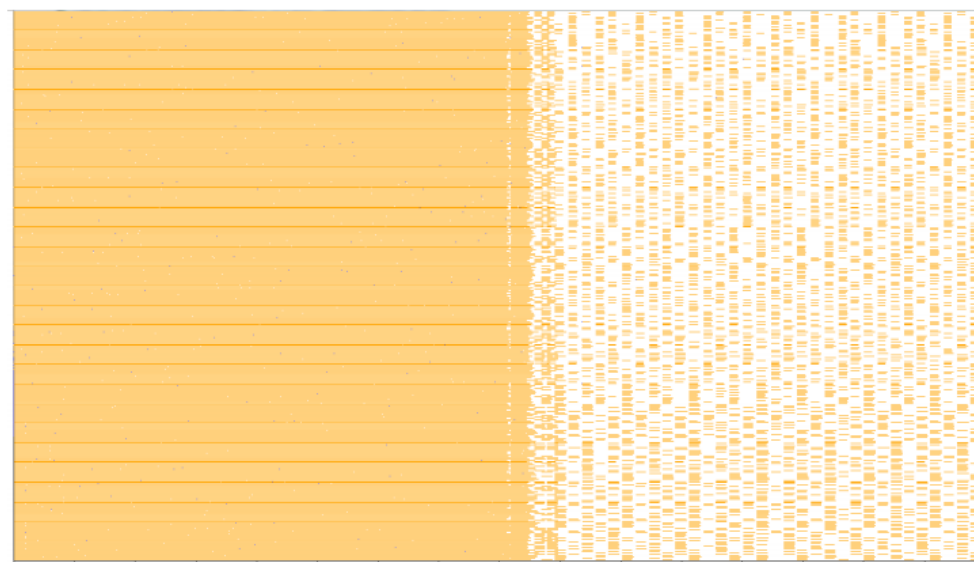

Figure 18: Several jobs run concurrently and saturate the cluster. Then all but one finish, and the one remaining runs in bursts of tasks, these bursts being not parallel enough to saturate the cluster (I recall it was a 480-core cluster and 160-task bursts).
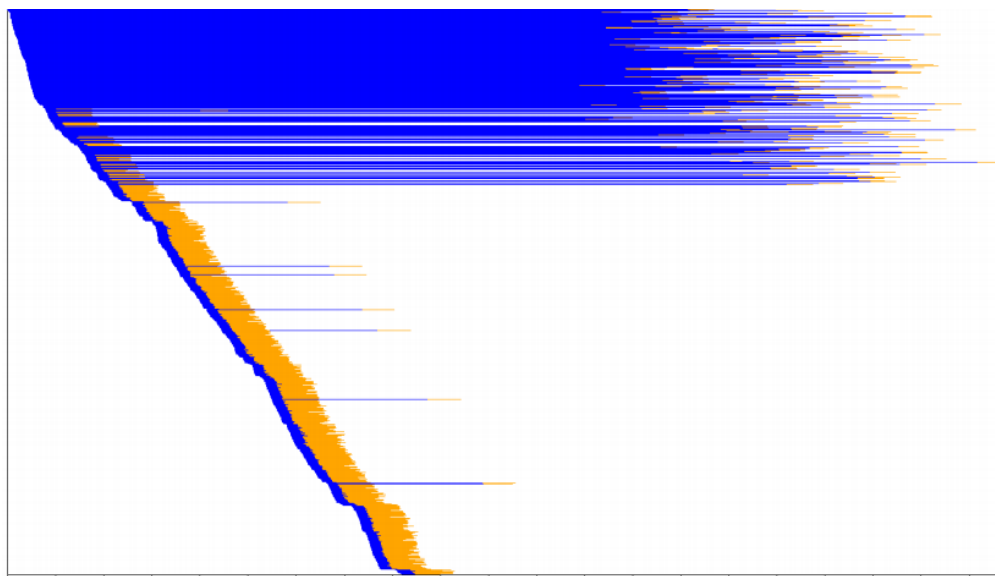
Figure 19: Orange: working, blue: fetching from memcached. We see that early calls to memcached take a ridiculous amount of time. Later calls sometimes take long too, but not that long (I recall the reason was a broken retry mechanism).

4   Acknowledgements

The following people contributed code to the tools or provided help otherwise at different times.

- Tim Docker wrote the excellent **Chart** library, on which **timeplot** is based.

- Ivan Tarasov promoted the tools, hosted an event where I could talk about them, and made a fork with a quantile graph with a log scale.

- Arnaud Bailly found a large number of important bugs.

- Dmitry Astapov and Julia Astakhova gave a lot of feedback.

- Orion Jankowski did a pullrequest which exposed a number of bugs.

- Jason Dusek, Ilya Teterin, Julia Chertkova gave examples of their usage of **timeplot**.

- Dmitry Popov provided the video encoding logs dataset.
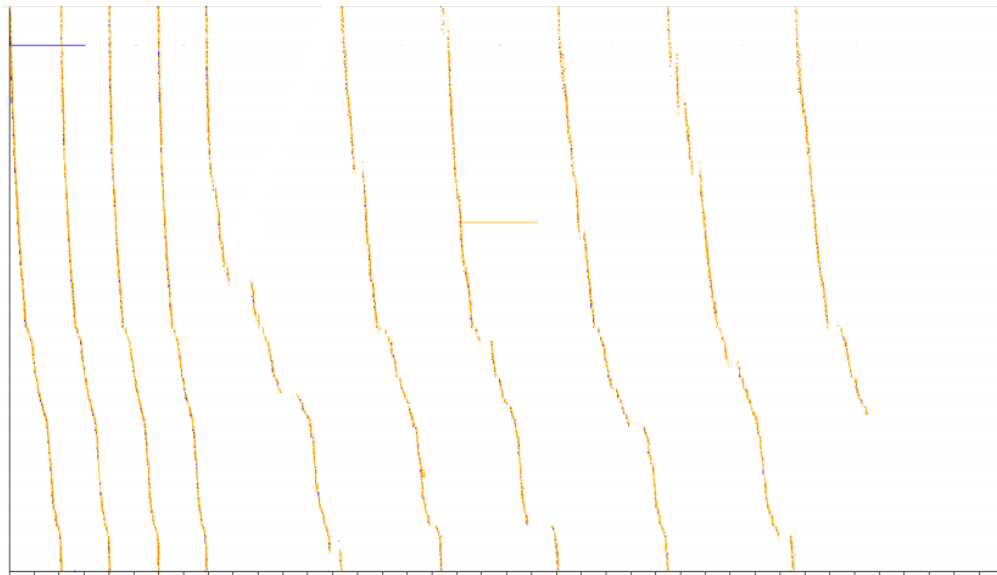
- Paul Graphov provided the DrWeb logs dataset.

Figure 20: There are 1900 tracks (cluster cores) here. The tasks that we put into the shared queue are by far too short, so the queue (and perhaps the task producer) becomes the bottleneck. We also see that the queue feeds tasks to workers in round-robin. And we also see that it slows down over time. And small pauses in the queue or task producer cost an awful amount of computing time because everyone is waiting for them.