Process-local load calculation

For Node.js

Load is measured using accumulated sliding averages of CPU clock ticks spent with the process in combination with number of queued/waiting events on the runloop.

Knowing how “loaded” a process have significance when for instance dealing with self-balancing networks or clusters.

Use-case scenario

Let’s imagine we have a host with 16 CPUs and we want to run a Node.js program on this host. We start 16 instances of the program, binding each process (program instance) to one specific CPU (affinity) so that each process runs on each CPU.

Our program keeps track of all program instances and their *load*. When a program instance is asked to perform a task it first checks its own *load*, and if the *load* is below a threshold value the program instance will accept the task and perform it. However, if the program instance’s *load* is too high the program will select a less loaded program instance and delegate the task to that instance.

Algorithm

- Keep accumulated values in process-local memory as V_{1-4}
- Let K be a constant natural number representing the resolution of V_4
- Let R be a positive time duration representing sampling interval
- Let X_1 be the result of (1 / exp(R / 1min))
- Let X_2 be the result of (1 / exp(R / 5min))
- Let X_3 be the result of (1 / exp(R / 15min))
- At R relative time interval perform the following:
  - Let T_1 be the number of CPU ticks spent with the calling process
  - If we have a previous value T_2 perform the following:
    - Let D be the result of $(T_1 - T_2)$ — number of CPU ticks since our previous sampling.
    - Let V_1 be the result of $((V_1 \times X_1) + (D \times (1 - X_1)))$
    - Let V_2 be the result of $((V_2 \times X_2) + (D \times (1 - X_2)))$
    - Let V_3 be the result of $((V_3 \times X_3) + (D \times (1 - X_3)))$
    - Let E be the number of events waiting to be processed on the runloop
    - Let V_4 be the result of $((V_4 \times X_2) + ((E \times K) \times (1 - X_2)))$
  - Let T_2 be the value of T_1 for use during next sampling

Implementation

The current implementation is sampling load every 5 seconds (R=5) by default with a event queue load resolution of 1000 (K=1000) and stores V_{1-4} as fixed-point integers (11 bit precision).

C++ internal API

m `static ev_timer node::load_timer`
— timer event used to schedule invocations of node::SampleLoad (.repeat=R).
m `static clock_t node::load_last_clock`
— holds the last registered CPU tick count (T_2)
m `static unsigned long node::load_accum[4]`
— accumulated values (V_{1-4}).
m `static int node::load_pendev_exp`
— waiting events exponent
m `static unsigned long node::load_norm`
— normal load updated by node::SetLoadSampleInterval.
m `static unsigned int node::load_exp_1min`
— 1 minute exponent updated by node::SetLoadSampleInterval (X_1).
m `static unsigned int node::load_exp_5min`
— 1 minute exponent updated by node::SetLoadSampleInterval (X_2).
m `static unsigned int node::load_exp_15min`
— 1 minute exponent updated by node::SetLoadSampleInterval (X_3).
f `static void node::SampleLoad(struct ev_loop*, ev_timer*, int)`
— take a sample of current load and update V_{1-4}.
f `static void node::SetLoadSampleInterval(ev_tstamp)`
— set sampling interval (and possibly start or stop node::load_timer).

JavaScript API

`process.load -> Array(4)` — read-only property which returns the current load values as floating-point precision numbers. [1min load avg, 5min load avg, 15min load avg, 5min queued evs avg]
`process.loadSampleInterval <-> uint32` — read-write property which controls how often process load is sampled and updated, in milliseconds. Setting this to 0 (zero) has the effect of disabling load sampling. Defaults to 5000.

Example

In this interactive session we illustrate how the load sampling rate can be read and manipulated as well as reading the load averages.

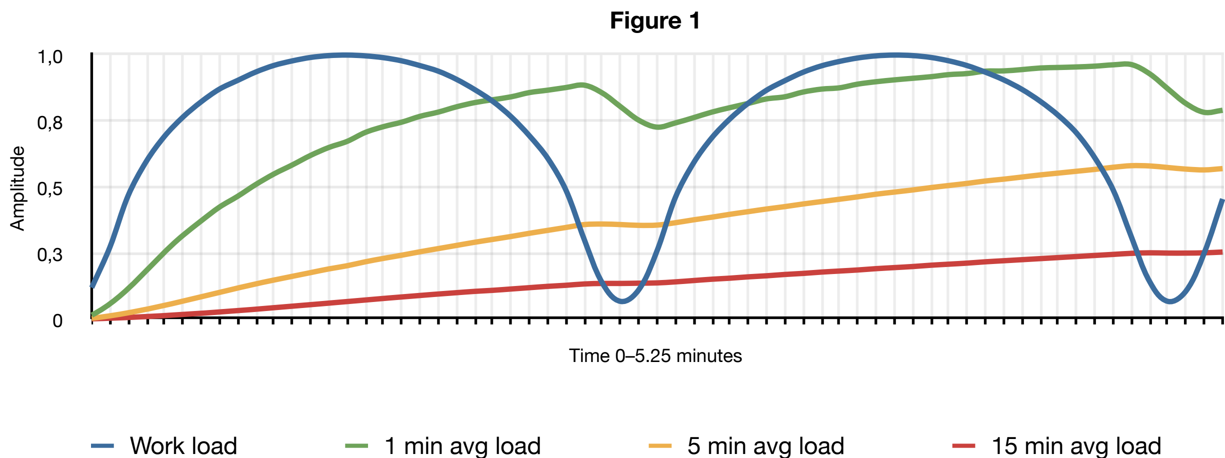
```
node> process.loadSampleInterval
5000
node> process.load
[ 0.0000354, 0.0000076, 0.0000024, 0 ]
node> process.loadSampleInterval = 1000
1000
node> setInterval(function(){ for(var c=100000;--c;){ Math.pow(123456, 123456); } }, 1);
{ repeat: 0, callback: [Function] }
node> process.load
[ 0.329499, 0.078633, 0.034458, 0.075 ]
node> ^C
```

Test and proof

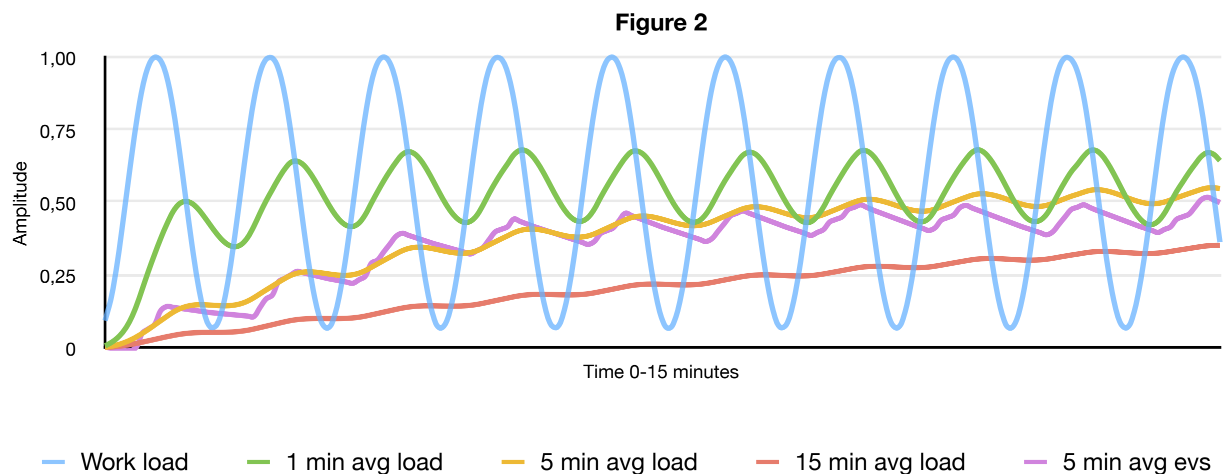
This simple test program generates load in a sine fashion (load pulsates over time):

```
var n = -(Math.PI/2), work = 0;
setInterval(function(){
  work = (Math.sin(n += 0.007) + (Math.PI-2)) / (Math.PI-1);
  for (var x,i=0;i<Math.round(work*4000);i++) {
    try { require('fs').readFileSync(__filename+'-'+i, 'utf8'); } catch(e) {}
    for (x=0;x<Math.round(work*4000)/2;x++) Math.sqrt(1234);
  }
}, 100);
setInterval(function(){
  console.log(work.toFixed(3)+', '+
    process.load.map(function(v){ return v.toFixed(3); }).join(', '));
},process.loadSampleInterval);
```

By running (modified versions of) this program and plotting its output we visualize how the load values behave over time during high but pulsating load.



In figure 1 the program have a average high load with low dips.



In figure 2 the program oscillates between high and low load, which eventually will bring the load values to about 50%. Notice how the 1 minute average is naturally lagging.