

FLP prototype Configuration benchmarks with *etcd*

The performance of *etcd* was evaluated as a potential backend for the Configuration module of the FLP prototype using representative workloads.

Revision date	Changes
April 26, 2016	Initial release

Workloads

At the start of a data-taking run, thousands of processes will need to retrieve configuration parameters from servers to be able to start processing, causing a large burst of GET requests. While the exact numbers regarding the amount of processes and the data volume are at this point unknown, CWG10 has made the following key estimations¹:

Metric	Value
Number of client processes	7k to 70k
Parameters per client process	100
Average size per parameter	100 bytes
Total configuration data volume	700 MB

Table 1: Key workload estimations

The structure of the data will likely vary greatly per client. Therefore, various data organizations were explored to see how *etcd* copes with them.

¹ Values taken from “CWG10 Software Requirements Specifications”

Name	Description	Query
Fragmented	Separate key-value pairs	One GET per parameter
Blob	Key-value pairs stored in single blob	One GET for whole blob
Flat	Key-value pairs grouped in one directory	Recursive GET
Tree	Key-value pairs grouped in nested directories, structured like a binary tree, with five parameters per directory	Recursive GET
TODO: Multiple trees	Mix between <i>fragmented</i> and <i>tree</i> : many small, separate trees	Multiple recursive GET

Table 2: Parameter organizations

Parameter key-values are numbered in a structured way so they can be verified client-side against the expected value. The values padded to 100 bytes. In addition, there might be overhead such as the space needed for the key string or supporting structures. For example, one key-value pair in a *blob* structure might look like: “key24=value[93 zeros]24”.

Test setup

Server hosts

Each server has one *etcd* instance running. Port 2379 must be opened for communicating with clients, 2380 for peers (in clusters). The *etcd* version used was 2.2.5, with the v2 API.

Benchmarks were ran in two modes: single server and clustered. In single server mode, *etcd* does not have to communicate with other instances, which might impact performance.

Starting an *etcd* instance is simple, and running as a simply requires some additional command line arguments. See <https://github.com/coreos/etcd/blob/master/Documentation/clustering.md> for a basic working example.

Client hosts

On the client side, a script is used that creates a number of instances of the benchmark program (typically hundreds to thousands). This script is scheduled using the “at” command to run at the start of a minute.

The benchmark program waits until the minute-and-10-seconds mark, then starts pulling parameters from the server. This wait simulates the “start signal” at the start of a data-taking run, and typically all programs start pulling within ~0.1 second.

The program has the following command line options:

- Output directory path

- Log ID: an ID unique to each run and client host. Current format: “result_[timestamp]_[hostname]”
- Server URIs. If multiple are given, it will choose only one to use. Currently, the choice is made pseudorandomly, based on PID modulo the amount of URIs.
- Number and structure of parameters. Format is [amount of parameters][mode]. Possible structures are 's' for *fragmented*; 'c' for *blob*; 'f' for *flat*; and 't' for *tree*². For example: 100t = 100 parameters in *tree* structure. For details about the structures, refer to Table 2.
- A debug mode. Left empty for normal benchmarking. May give “put” for putting parameters instead of getting, or “skipwait” to skip the wait to the minute-and-10-seconds mark

Network timeouts and errors are logged to a file per PID. If too many errors have occurred (such as exceeding a maximum amount of reconnect retries), it will stop.

If the program is able to successfully get all the parameters from the server, a line with the format “log ID, process ID, start time, end time, difference between start and end” is appended to a file called “result_[log ID].csv”. The file is opened in append mode and buffering is disabled to make sure it doesn't become garbled (unbuffered appends are/should be atomic). These are the files that are retrieved from the client hosts. They can be concatenated and loaded into a spreadsheet program for analysis.

Deployment

The benchmark was deployed using an assortment of bash scripts, mostly based around the Parallel-SSH utilities (<https://code.google.com/archive/p/parallel-ssh/>). Some example tasks that are done this way, in roughly the “order” of usage:

- Copying executables to hosts
- Adding firewall rules
- Putting parameters into an *etcd* server
- Starting *etcd* servers
- Scheduling runs
- Retrieving results from client hosts
- Canceling pending runs
- Killing server processes
- Clean up files on hosts, such as logs & results

² The odd naming is due to historical reasons, as the structures used to have different names.

Runs are scheduled using the “at” command. They are given a timestamp which can later be used to retrieve the results from the client hosts. Multiple runs can be scheduled at once to different points in time, allowing for a coffee break while the benchmarks are executed.

Machines

The following machines have been used in the benchmarks:

Hostname	Specifications
aidrefpc001/45	2x E5520, 8 threads, 2.26 GHz
aidrefsrv10	2x E5-2650 v2, 16 threads, 2.6 GHz
aido2qc10/13 aido2qc40/43 aido2web1	2x E5-2640 v3, 16 threads, 2.6 GHz

Table 3: Machine specifications

Single server results

For the single server benchmarks, the following machines were used:

Role	Hostname
Server	aidrefsrv10
Clients	aidrefpc001/045

Time to pull parameters, fragmented structure

1 server, 43 client hosts, 16 processes per client

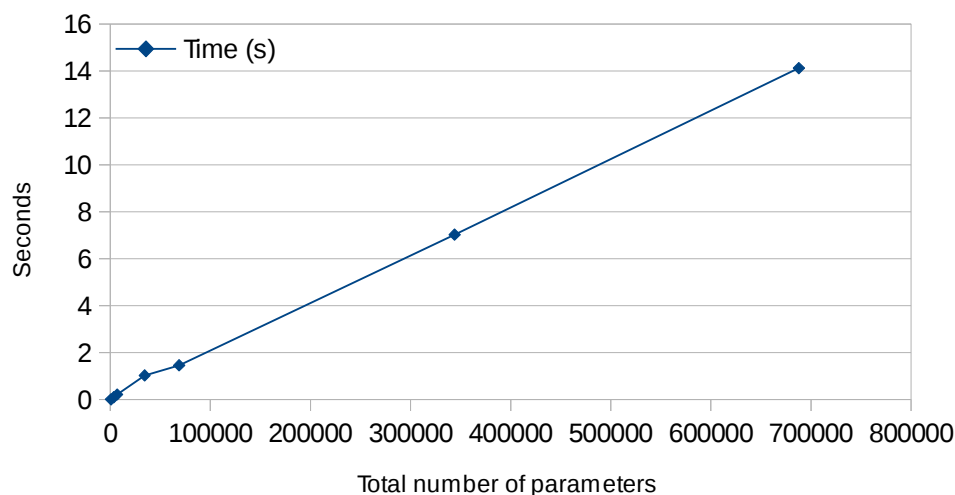


Chart 1

It quickly became clear that using one query per parameter is very detrimental to performance. Compared to the speeds achieved with the other techniques, this is quite slow. For example: the highest point in Chart 1, 688k parameters in 14.1 seconds, compared to Chart 2's "100 parameter per process" series that gives 12.1 million parameters in 11.4 seconds.

Time to pull parameters, blob structure

1 server, 42 client hosts, 16 processes per client

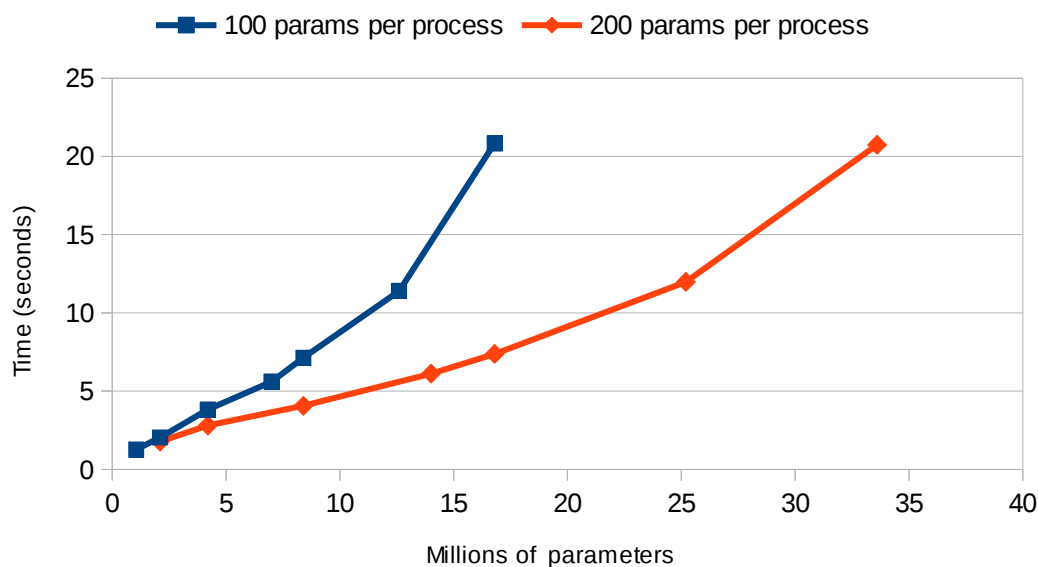


Chart 2

It can also be seen in Chart 2 that with *blob* parameters, doubling from 100 parameters per process to 200 seems to have little effect on performance: the data points from the two lines are very closely aligned on the y-axis.

This is not the case for the *flat* and *tree* organizations, as can be seen in Chart 3.

Time to pull parameters, varying structures

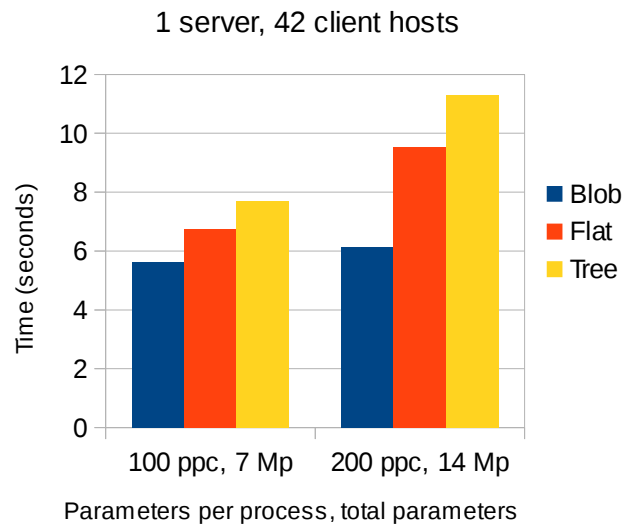


Chart 3

These differences can be explained by the additional complexity of the *flat* and *tree* structures. The *blob* requires one key lookup; the *flat* requires many lookups in one directory; and the *tree* requires many lookups in multiple directories.

In addition, the JSON data the server responds with can carry a significant amount of overhead. In Table 4 are given a few typical structures and the size of the corresponding JSON data.

Parameter structure	Usable size (bytes)	JSON response size (bytes)	Overhead
100 blob	10000	10892	8.92 %
100 flat	10000	18098	81.0 %
100 tree	10000	21214	112 %
200 blob	20000	21796	8.92 %
200 flat	20000	36198	81.0 %
200 tree	20000	44214	121 %

Table 4: JSON response size of typical parameter structures

The *flat* and *tree* structures are the main offenders: the size of “100 tree” is close to that of “200 blob”, despite containing half the parameters. And the overhead of “200 tree” is even higher than “100 tree”. This is because for each key-value pair, the JSON contains:

- The key, with full absolute path (not relative to parent directory)
- Value
- The “modifiedIndex”
- The “createdIndex”

See Table 5 for an example snippet. Directories get a similar treatment.

Since only one key-value pair is needed for the *blob*, that saves a lot of bytes.

```
{
  "action": "get",
  "node": {
    "key": "/test/tree100",
    "dir": true,
    "nodes": [
      {
        "key": "/test/tree100/key3",
        "value": "value00[...]03",
        "modifiedIndex": 103,
        "createdIndex": 103
      },
      {
        "key": "/test/tree100/key4",
        "value": "value00[...]04",
        "modifiedIndex": 104,
        "createdIndex": 104
      },
      {
        "key": "/test/tree100/dirA",
        "dir": true,
        "nodes": [
          {
            "key": "/test/tree100/dirA/key8",
            "value": "value00[...]08",
            "modifiedIndex": 78,
            "createdIndex": 78
          }
        ]
      }
    ]
  }
}
```

Table 5: Snippet of a “100 tree” JSON response

The speed differences can likely be attributed to more key-value fetching, more JSON writing, and longer network transfers. Especially in bandwidth-limited scenarios, this could pose a real issue.

Another thing to keep in mind, is that in these benchmarks it is assumed that parameter values are 100 bytes. It could be that this is pessimistic. If it were smaller, the relative overhead of the *flat* and, especially, the *tree* structures would grow compared to the *blob* structure.

The v3 API might improve matters here: <https://coreos.com/etcd/docs/latest/rfc/v3api.html>. It uses Google's Protocol Buffers to serialize data into a binary format, which could reduce overhead.

Clustered results

For the clustered benchmarks, the following machines were used:

Role	Hostname
Servers	aido2qc10/13
	aido2qc40/43
	aido2web1
Clients	aidrefpc001/045

Using multiple servers improved the read performance significantly (Chart 4).

Time to pull parameters, various data organizations and cluster sizes

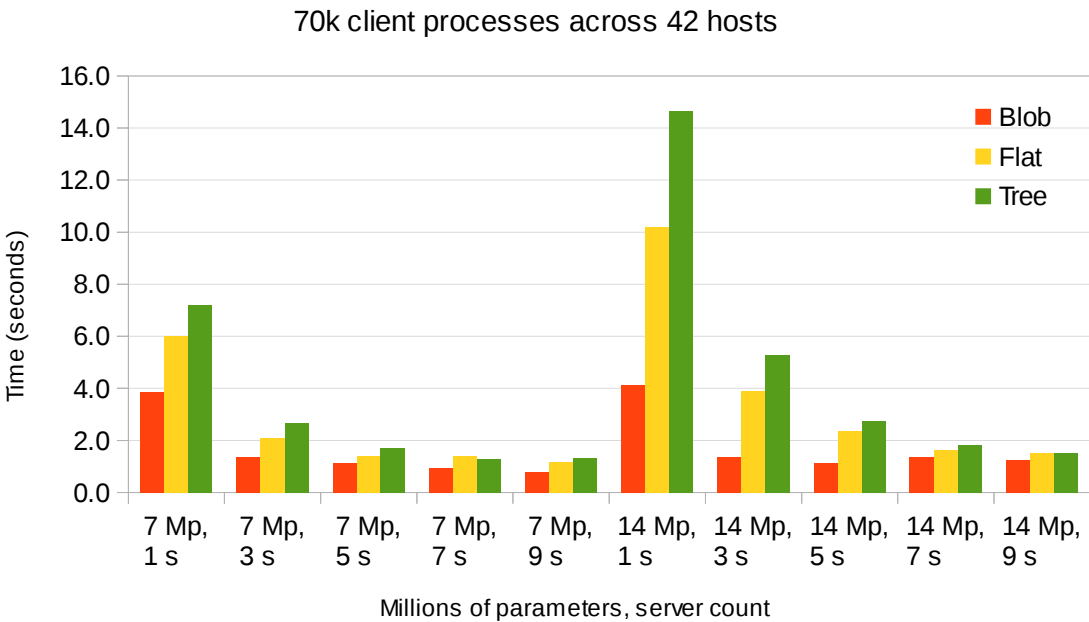


Chart 4

Performance gains are roughly linear, as can be seen when plotting the time as a function of the reciprocal of the amount of servers (Chart 5).

Time to pull 7M parameters as a function of 1/servers

70k client processes across 42 hosts

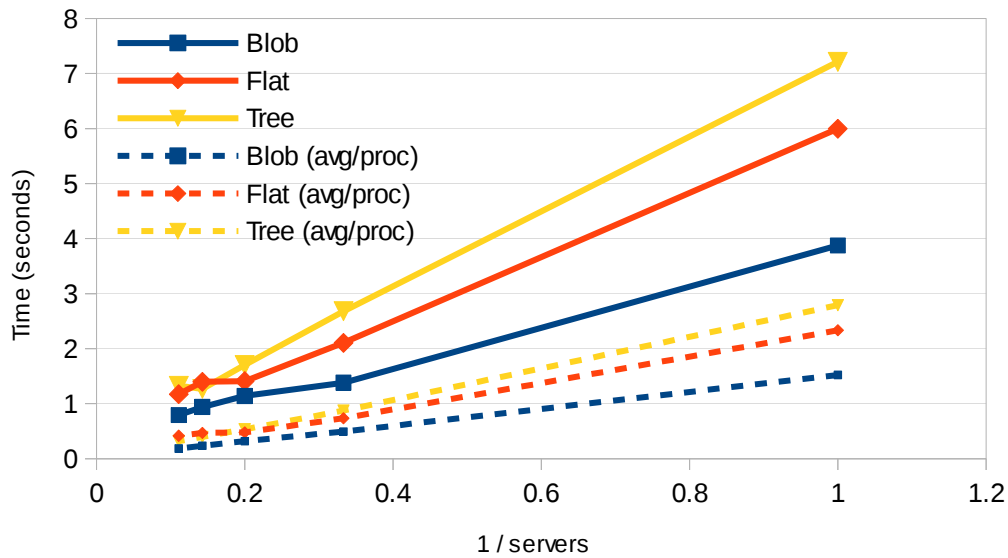


Chart 5

Server resource usage

Chart 6 through Chart 9 show CPU and network usage measurements taken from a server during benchmark runs. All of them show that usage is clearly not ideal, even though ultimately the performance is acceptable (aside from with the *fragmented* structure).

For the 9 server cluster setup (Chart 6 and Chart 7), CPU usage and transmission rates max out at the start; then dip to below 30% for a significant time; before finally increasing again.

The single server setup in Chart 8 shows spiking and an overall downwards trend.

In Chart 9 the *fragmented* structure was used which showed spiking, low transmission rates and no trend.

The causes behind these effects are unknown.

Resource usage

9 servers, 42 client hosts, 70k processes, 14M parameters in "200 tree" structures

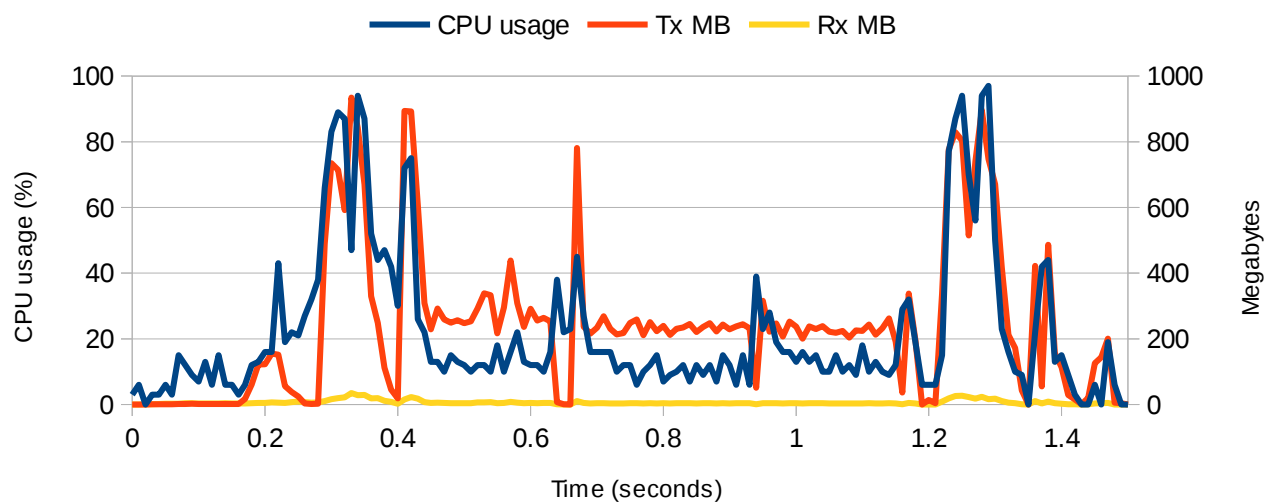


Chart 6

Resource usage "200 blob"

9 servers, 42 client hosts, 70k processes, 14M parameters

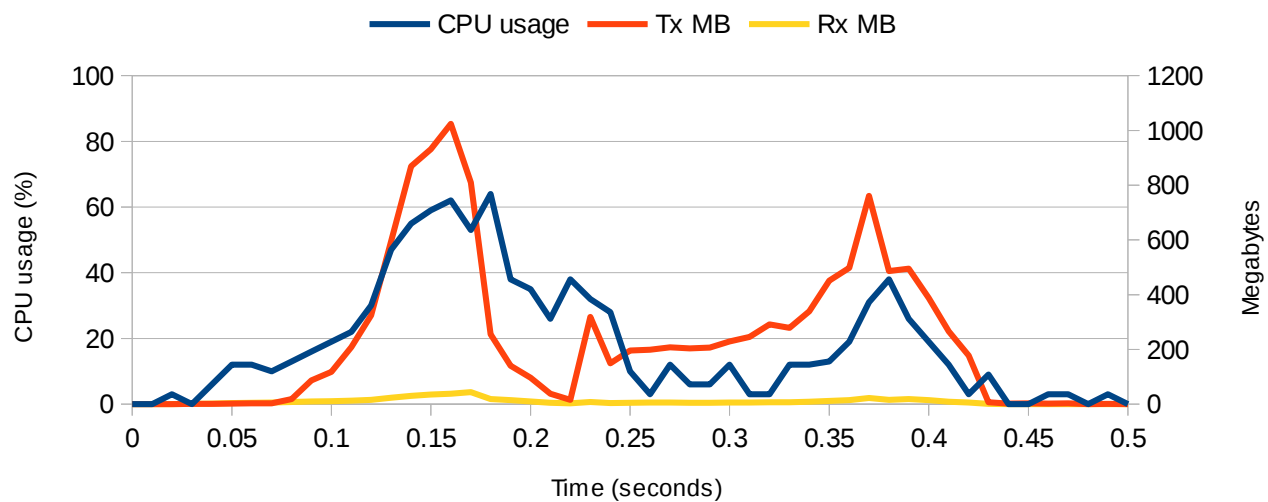


Chart 7

Resource usage "200 tree"

1 server, 42 client hosts, 70k processes, 14M parameters

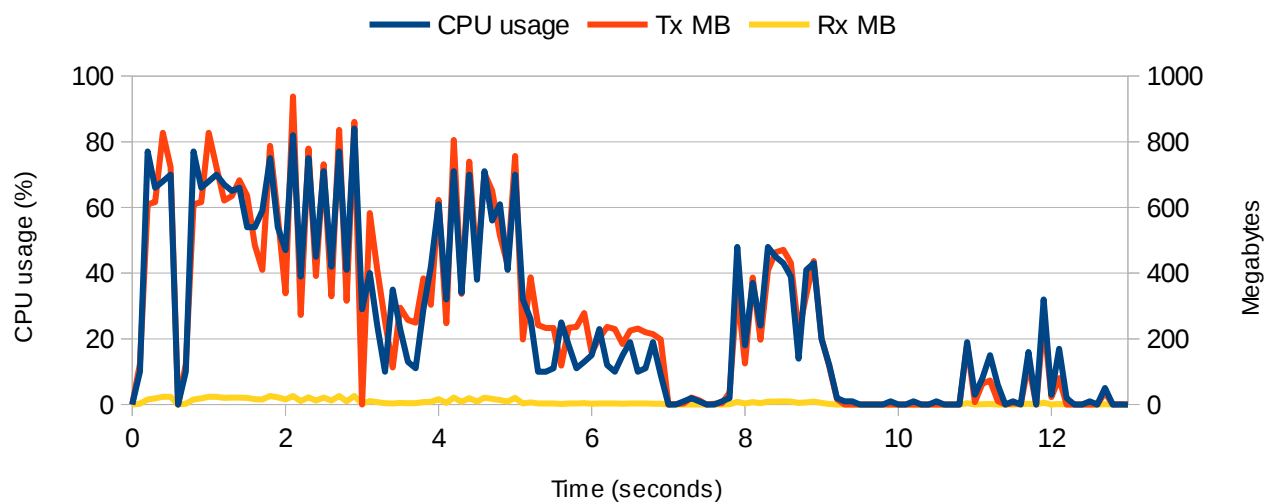


Chart 8

Resource usage "50 fragmented"

9 servers, 42 client hosts, 70k processes, 3.5M parameters

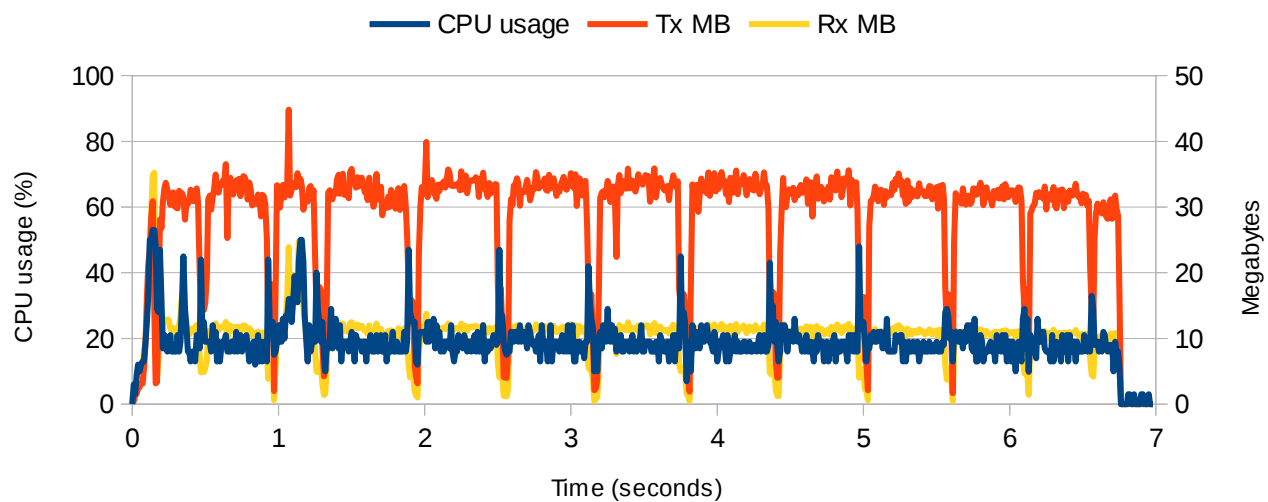


Chart 9

Appendix A – Notes

Libcurl timeouts

The *etcd* API used in these benchmarks uses libcurl in the back end. By default, libcurl uses timeouts and an exponential backoff algorithm. Because the server gets flooded with requests from thousands of processes, timeouts may occur, predominantly at the start. This manifested as large, exponentially spaced spikes in traffic. To get around this a fixed timeout was used. It may be more optimal if the timeout is more adaptive. The fixed timeout requires adjustment, depending on the server load. If the timeout is too low, the server is swamped and throughput suffers, at least initially.

Sub-second timeout in libcurl requires a workaround on Unix. Set `CURLOPT_NOSIGNAL` to 1. More info: <http://php.net/manual/en/function.curl-setopt.php#104597>

Frequency of amount of timeouts per process

7M parameters in "200 tree" structure, 42 clients, 70k processes, 10s timeout

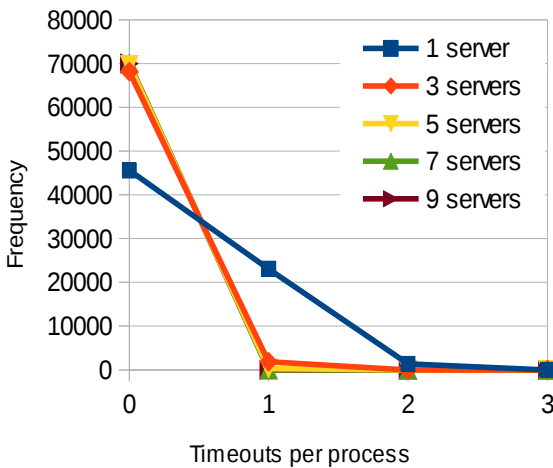


Chart 11

Frequency of amount of timeouts per process

9 servers, 42 clients, 70k processes, 250ms timeout

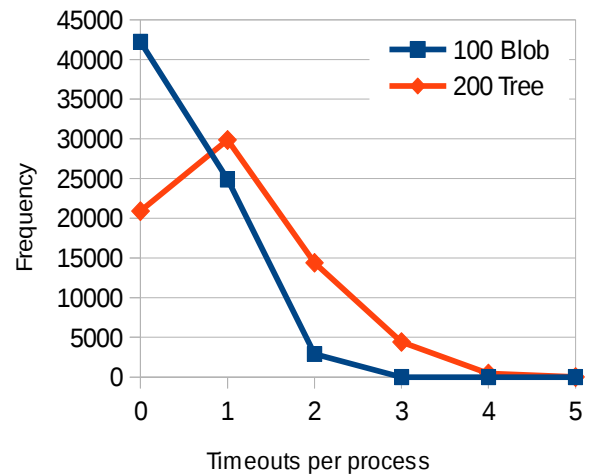


Chart 10

In Chart 10 and Chart 11 are the results of some measurements of the frequency of timeouts that occur per process during a run. In Chart 10, with the 10 second timeout, the frequency drops off quickly when 3 or more servers are used. In Chart 11, a more aggressive 250 millisecond timeout is used, and it can be seen that the timeouts are more prevalent for the more demanding “200 tree” structure. With a longer timeout length and higher server count (as in Chart 10), timeouts are not a significant issue.

Open files limit

Make sure that the max amount of open files is not too low on the server. Because every TCP connection uses a file descriptor, many will be used to service the requests from client processes.

Hitting the open files limit will lead to a lot of connection timeouts and very bad performance. Use “ulimit -n” to check, and for example “ulimit -n 65535” to set.

Monitoring

For manual, real-time monitoring, “htop” was used for CPU usage and “nload” for network throughput. Bandwidth did not appear to be a bottleneck in the benchmarks.

More granular logging of metrics for chapter “Error: Reference source not found” was done with “collectl”.

Capturing packets

When using tcpdump to capture packets for inspection, the burst of network activity requires a large buffer size, or many packets will be dropped. For example: “tcpdump -B 100000 -i eth0 -w tcpdump.cap”

DNS avoidance

Use IP addresses instead of hostnames, to prevent DNS overload. Symptoms include very low performance during a run; not being able to SSH into machines after a run.

Instance names & data directories

Each instance of *etcd* must have a unique name if it is running in a cluster. By default, the instance name is also used for the data directory, where *etcd* stores configuration data and its database. If this directory is not cleaned up between runs, and the same name is reused between different cluster configurations, *etcd* can get confused, thinking it should connect to an older cluster with different tokens. To avoid this, different names were used to only for different instances (which is required), but also for different configurations. For example, the third server in a cluster of size 9 is named: “cls9_srv3”.