



Home

Scale up your monitoring with Supermon

Matt Sottile

Ron Minnich

Los Alamos National Laboratory

Supermon is a flexible set of tools for monitoring clusters at very high data rates. On a Pentium III/800 system, for example, Supermon can extract self-describing data from the Linux kernel at up to 6,000 samples per second. That data rate is as fast as the `sysctl` interface, and is over 100 times faster than the current-generation interfaces accessing the `/proc` virtual file system.

A monitoring system's performance is crucial. Using Supermon, we were able to observe behavior in systems such as **MPI** that had never before been seen, and have even uncovered bugs in library code. In addition, while Supermon presents very low system overhead, it extracts more performance information than current-generation monitoring tools do. Tools, such as `xload`, `xmeter`, and other `rpc.rstatd`-based utilities, produce adverse impact on applications running on the cluster nodes at 10 samples per second (10 Hz), and do not present enough information to be usable.

Supermon is used primarily on Linux-based systems at the moment with ports to other operating systems possible. The kernel module that provides data for single cluster nodes to `mon` is the only Linux-specific component. Porting Supermon for a different operating system would involve the following two coding tasks: First, the equivalent of the kernel module would have to be written to extract all data to be monitored and provide it to `mon` as a set of `s`-expressions. Alternatively, existing libraries can be used (e.g. `kvmlib`) to extract the data from the kernel. Second, depending on the method by which this data is made available, `mon` may require modification to point at the proper data source for sampling. An older Supermon version based on `rstatd` could be modified to turn `rstat` data into `s`-expressions compatible with the current architecture. All other portions of Supermon are portable due to the ASCII-based `s`-expression protocol and minimal use of operating system-specific features in most of the tools.

Supermon elements

Supermon consists of three distinct components (see Figure 1): (1) a loadable kernel module providing data, (2) a single node data server (`mon`), and (3) a data concentrator (Supermon) that composes samples from many nodes into a single data sample. The kernel module provides data samples through an entry in the Linux `/proc` filesystem. The single node server and data concentrator

User login

Username: *

Password: *

Log in

- [Request new password](#)

Navigation

- [Doctoral Symposium 2009](#)
- ▶ [News aggregator](#)

Syndicate



allow clients to retrieve data samples through TCP sockets. Clients that wish to use that data must connect to any of those components, and parse the data into their preferred format. All three components speak the same protocol, which is based on *symbolic expressions (s-expressions)* originally introduced as part of the LISP programming language.

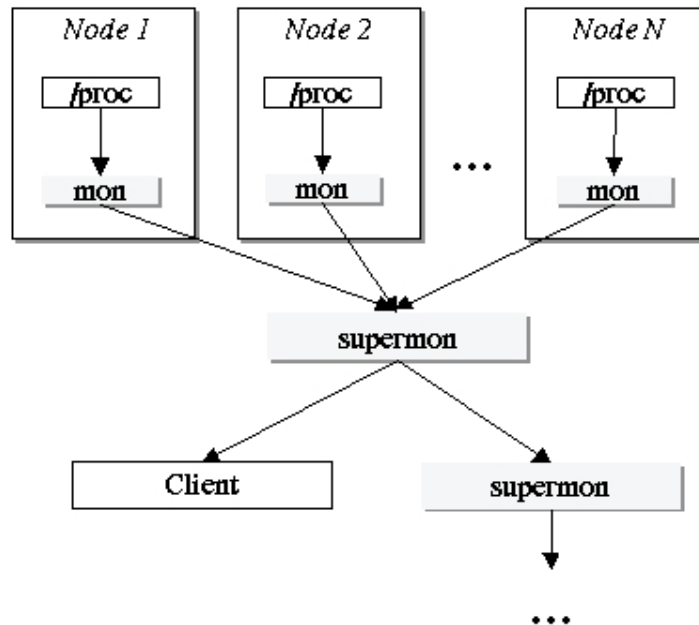


Figure 1.
Supermon's main components

Communication protocol

Supermon defines a client-server protocol. Protocol packets consist of symbolic expressions. Unlike SunRPC packets, s-expressions contain self-describing data. None of the RPC compiler or XDR tools are needed by the protocol. The simple, recursive form of s-expressions allows them to encode arbitrarily complex data.

Other useful properties of s-expressions: They are neither fixed-size, nor are they binary data. Standard RPC packets, in order to achieve computer architecture-independence, must strictly follow a binary format and size. In contrast, s-expressions can vary in size as needed, and achieve architecture independence by not relying on binary data.

We put the power of s-expressions to good use in Supermon. Perhaps the most interesting aspect of the Supermon protocol is that it is *composable*: Supermon clients can serve as Supermon servers. An individual Supermon server can act as a client and aggregate the s-expression streams from multiple Supermon servers. Those servers, in turn, can also aggregate other streams. Standard RPC servers, such as automounters, allow limited composition, but their aggregation capability is very limited and involves information loss from the initial server to the final client. The Supermon protocol supports aggregation with no loss of information.

The Supermon protocol was designed to be used on very large clusters, and scalability and composability were important requirements. Thus, Supermon employs s-expressions at all levels: From the kernel module that provides the initial data, to client applications using that data. Supermon also provides a compact and efficient s-expression parser.

The kernel module

The kernel module is a dynamically loaded module that inserts an entry into the `/proc/sys` tree in the Linux kernel. The entry is a directory named `supermon` with two nodes: `s` and `#`.

The `s` entry returns Supermon data in s-expression form (see Figure 2). Data are grouped into categories. Each category contains a number of named fields with their values. There can be more than one set of values for named fields, as can be seen in the `netinfo` entry in the example. The machine in the example has three Ethernet interfaces: `lo`, `eth0`, and `eth1`. Consequently, each named field has three elements, one for each interface. The first component of the `netinfo` list is the set of names of the interfaces. On a laptop, interfaces come and go as cards are plugged in. Supermon data will reflect that change: as the interfaces appear and disappear, the size of the list changes. Supermon data can be dynamic in a way that is difficult or impossible for `sysctl` entries to match.

```
(cpuinfo
  (user 232007070)
  (nice 1314934)
  (system 0)
)
(avenrun (avenrun0 2060) (avenrun1 2056) (avenrun2 2048))
(paging (pgpgin 16) (pgpgout 0) (pswpin 0) (pswpout 0))
(switch (switch 549615))
(time (timestamp 0xec05d78898) (jiffies 0xebd2e4b))
(netinfo
  (name lo eth0 eth1)
  (rxbytes 0 0 45681848671)
  (rxpackets 0 0 31522281)
  (rxerrs 0 3 0)
  (rxdrop 0 0 0)
  ...
)
```

Figure 2.

Supermon output from the kernel module for the `s` command

The `#` entry returns data descriptors, also in s-expression format (see Figure 3). All categories follow an identical format: category name (e.g., `cpuinfo`), cardinality of the category (`(nr 1)`), and the field names for the category (e.g., `user`). In the example, the `cpuinfo` field has a cardinality of 1; on an SMP with 2 or 4 CPUs, it has a cardinality of 2 or 4. The `netinfo` category has a cardinality of 3, since this machine has three interfaces; again, that cardinality changes as interfaces come and go.

```
(cpuinfo (nr 1) (user nice system)
)
```

```
(avenrun (nr 1) (avenrun0 avenrun1 avenrun2))
(paging (nr 1) (pgpgin pgpgout pswpin pswpout))
(switch (nr 1) (switch))
(time (nr 1) (timestamp jiffies))
(netinfo (nr 3) (
  name rxbytes rxpackets rxerrs rxdrop rxfifo rxframe
  rxcompressed rxmulticast txbytes txpackets txerrs
  txdrop txfifo txcolls txcarrier txcompressed)
)
```

Figure 3.

Supermon output from the kernel module for the # command

Note that the data structure is self-describing, and is easily parsed by programs (or people). That output format is far more useful than the standard Linux `/proc` format entries (see Figure 4). The standard Linux `proc` entries do not share an identical format: `cpuinfo` is presented as name-value pairs separated by colons; `meminfo` is partly a table (first three lines) and partly name-value pairs (the rest of the lines). `slabinfo` is the most confusing: the first line at least does describe the origin of the data, i.e. the slab allocator, but the remaining lines present data with no clear meaning.

processor	:	0				
vendor_id	:	GenuineIntel				
cpu family	:	6				
model	:	8				
model name	:	Pentium III (Coppermine)				
Mem:	total:	used:	free:	shared:	buffers:	cached:
Swap:	262381568	233357312	29024256	0	25935872	78348288
MemTotal:	479473664	569344	478904320			
MemFree:	256232 kB					
MemShared:	28344 kB					
	0 kB					
slabinfo - version: 1.1 (statistics)						
kmem_cache	54	58	136	2	2	1 : 54 54 2 0 0
pio_request	0	0				
tcp_tw_bucket	1	42				

Figure 4.

Three different Linux `/proc` entries: `cpuinfo`, `meminfo`, and `slabinfo`

We do not currently provide `swapinfo` or `meminfo` statistics: They have proven far too costly to query at speed. We describe those problems in a [previous work](#).

Mon and Supermon

Two small server programs move data from the kernel to clients, and provide that data via TCP at both single and multiple node levels. At a single node, a kernel module provides data in its two `/proc` entries (see above). The *mon* server acts as a filter between `/proc` and the TCP clients: It parses the `s-`expressions found in `/proc`, adds a minimal amount of information, and passes that data to clients on demand. For each client that connects to it, *mon*

maintains a bitmask reflecting the data fields that particular client requests in a sample. That way, mon filters data and reduces wasteful network traffic.

A second server - *Supermon* - lets clients see a snapshot of a *set of nodes* in each sample. Supermon connects to nodes that run mon servers, and concentrates their data. It then presents the data sampled from many mon servers in a single data sample. The data format provided to clients by Supermon is identical to mon's data format. That allows many Supermon servers to be created, each sampling from a subset of the nodes within a cluster. New Supermon servers could then be started to connect to the Supermon servers already monitoring portions of the cluster.

Hierarchical Supermon servers improve performance in situations where a cluster has many nodes and sampling rates are high. Supermon provides a bitmask-based filter for each client (similar to mon), which is then used to improve efficiency between the Supermon/mon and Supermon/client connections.

Performance

The majority of development time on Supermon was dedicated to making each portion of the system as efficient as possible. Efficiency at each level allows Supermon to achieve the high sampling-rate goal that was unattainable with older monitoring tools.

The main metric for measuring Supermon's capabilities is the number of samples per second that can be taken from a node or cluster. We are most interested in the lower bound for sampling rates when samples contain the maximum amount of data. All of our benchmarks were performed using the complete data set offered by mon from each node. (Note: That data set is larger than what rstatd provides).

The system we used for benchmarking was the LANL ACL xed cluster, which is composed of three types of Compaq Alpha-based compute nodes. There are a total of 124 nodes, containing 152 processors with a total of 200GB of memory. The specifications of the nodes are given in Figure 5.

<i>Node type</i>	<i>CPUs</i>	<i>Memory</i>	<i>Number</i>
DS-10	1	1GB	104
CS-20	2	2GB	16
ES-40	4	16GB	4
<i>Total:</i>	152	200GB	124

Figure 5.

The LANL ACL xed testbed cluster.

The benchmarks were run from the front-end node - an ES-40 used for starting jobs and controlling compute nodes. Jobs were issued to compute nodes and managed by **BProc** (Beowulf Distributed Process Space) that provides a single-system image with respect to processes. Each benchmark attempted to read Supermon data samples as many times per second as

possible, slowly working from a small number of samples up until the duration of sampling took longer than one second. Each sample contained all possible data provided by Supermon to reflect worst-case performance. (In general, applications request only a subset of the data, and are capable of higher sampling rates than those we report.) Figure 6 shows approximate data packet sizes, with about 5-10 bytes variation per sample depending on the data values transported.

<i>Benchmark</i>	<i>Data size</i>	<i>Nodes</i>
Kernel	~ 800 bytes	1
Mon	~ 950 bytes	1
Supermon	~(950 * N) bytes	N

Figure 6.

Approximate data sizes for each benchmark.

Kernel module performance

The first test measures a program's maximum sampling rate reading directly from /proc entries provided by the Supermon kernel module. It compares the performance of that Supermon capability to the method used by RPC-based rstatd to gather its data. The lowest sampling rates we found from /proc were 3,400Hz on the DS-10 and CS-20 nodes, while the ES-40 nodes achieved 6,000Hz. The test was also run on an Intel Pentium III machine, with performance comparable to the ES-40.

Comparing those results to rstat's `get_stats()` call, we found a huge performance improvement (see Figure 7). Using the same benchmark program used for measuring /proc - with a minor change to call `get_stats()` instead of reading a file - we observe a peak performance of 300Hz. Not only is that an order of magnitude slower than /proc, we will shortly see that that is slower than the sampling rates observed after the data has passed through a single mon process and a single Supermon process.

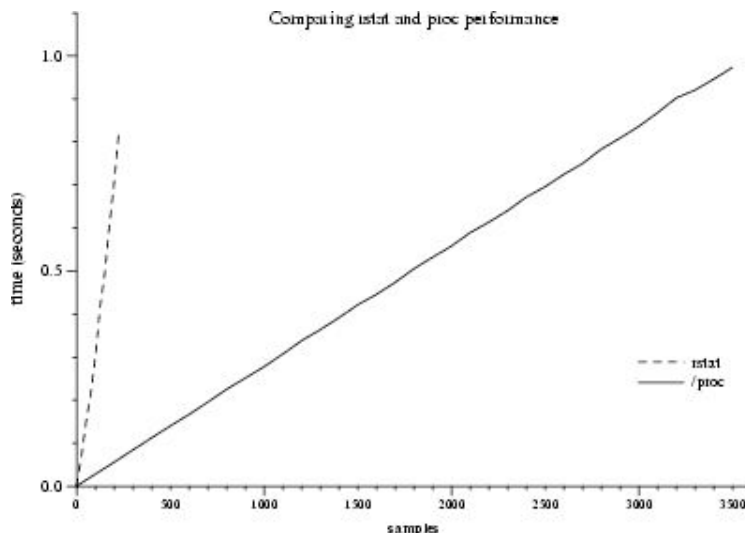


Figure 7.

Number of samples over time for /proc and rstat. /proc achieves a peak sampling rate of 3,500 Hz while rstat can only achieve 300 Hz.

Mon and Supermon performance

To measure the performance of mon and Supermon, we used a program similar to the one for measuring /proc. Instead of opening a file handle and reading, the benchmark program opened a socket to the data server, sent a command asking for all data provided by the server, and read that data before sending another request. With mon, the data must pass over two channels: It is first read from the /proc entry, and then sent between mon and the client over TCP.

For Supermon, using the same benchmark as for mon (which was possible, since mon and Supermon use an identical protocol), we measured the maximum sampling rate for various configurations of Supermon. The first case had Supermon gather data from a single mon process (see Figure 8). The second case measured multiple nodes: it tested with 5, 10, 20, and 100 nodes monitored by a single Supermon process. The final test gauged Supermon's performance in a hierarchical topology. There were two cases: First, each Supermon had a fanout of 10 nodes; in the second case, each Supermon monitored 50 nodes. At the hierarchy's root a single Supermon process gathered the entire cluster data set from the Supermons observing subsets of nodes.

To make the test environment closer to actual practice, we took care to lay the Supermon servers out such that each ran on the first compute node in each node subset. For example, if there were 100 nodes in groups of 10, a Supermon server would run on nodes 0, 10, 20, etc. The Supermon server responsible for data gathering from each node subset ran on a computer outside of the subset. Finally, the client ran on the cluster front end. Two reasons warranted that separation of Supermons: First, we wanted to avoid overwhelming a single machine with many Supermon servers exchanging data. Second, we wanted to avoid any effects caused by loopback devices or TCP optimizations for socket communication within a single computer that could generate results showing higher sampling rates, but without accounting for the network's effect on the monitoring process.

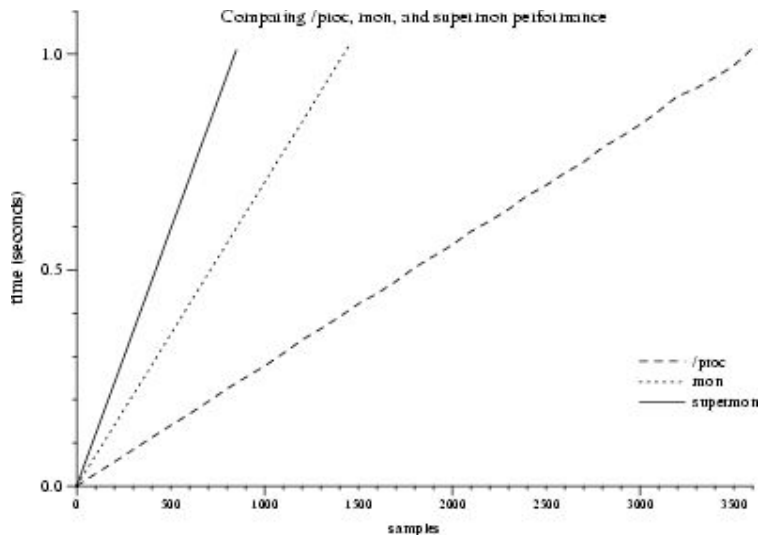


Figure 8.

Number of samples over time, comparing sampling rates from /proc, mon, and Supermon. Decrease in maximum sampling rate is due to the network.

Figure 8 shows the performance results when testing Supermon's scalability. The number of nodes corresponds to the number of mon servers a single Supermon connects to. When all 100 DS-10 nodes were monitored, we performed three different tests to show the effect of hierarchical Supermon servers on performance. The basic case followed a flat topology: A single Supermon connected to all 100 nodes. Next, each Supermon connected to 10 clients (a single Supermon connected to 10 other Supermons, each of which then connected to 10 mons). Finally, a single Supermon connected to two Supermons responsible for monitoring half of the cluster each (50 mons). We were surprised to find that, contrary to popular belief, hierarchy is not guaranteed to increase performance. In our case we showed that the additional network traffic between the layers of Supermon servers had a negative impact on the sampling rates.

<i>Nodes</i>	<i>Sampling rate</i>
5	400Hz
10	225Hz
20	125Hz
100 (Flat)	66Hz
100 (10-node fanout)	57Hz
100 (50-node fanout)	35Hz

Figure 9.

Scaling results for Supermon.

Conclusions and Future Work

We found S-expressions to be an ideal interface for getting information from the kernel. S-expressions' structure adapts to changing kernel resources well (e.g. PCMCIA cards), something no existing Linux interface supports. We feel

that all existing Linux /proc interfaces should be redone using S-expressions, replacing the current large number of inconsistent output formats.

Supermon also shows that RPC-based protocols - such as SunRPC - are neither necessary nor sufficient for this type of monitoring. They are not necessary as they have no performance advantage over the textual Supermon protocol. They are not sufficient as they do not handle variable-sized data well, and worse, require complex overhead at each end for converting data between architecture types. The textual S-expression format is inherently architecture-independent and just as efficient. It is long past time to retire SunRPC for this sort of use and to eliminate the rpc.rstatd daemons as well.

Our next steps for Supermon are to further test scaling. We are looking at placing intermediate daemons - *filtermons* - in the tree to aggregate the data so that a collection of many nodes would be presented as one average. We will be putting more hardware information into Supermon, in addition to the limited network hardware statistics we now have. We will also be tying Supermon into our scheduler, so that the scheduler can make scheduling decisions based on hardware availability.

In addition to improving the monitoring system, we will also be looking at techniques for analyzing monitoring data for failure prediction, algorithm analysis, and performance optimization. In keeping with the spirit of the monitoring framework, the analysis techniques will not only be looked at from an analytical perspective but in terms of their computational complexity. That focuses on the ability to produce results using high-speed data samples at runtime instead of post-mortem analysis of stored monitoring data.

Resource: [Supermon's Web site.](#)

[Standard documents for the Message Passing Interface \(MPI\)](#)

[MPI Forum's Web site.](#)

[sysctl man pages](#)

Resource: [Supermon's Web site.](#)

[More sysctl info](#)

[Home page for BProc \(Beowulf Distributed Process Space\)](#)

» [Login to post comments](#)

[viagra cialis levitra](#), by young773HK

[viagra uk, cialis levitra](#) by young773HK

[Bisexual threesome fuck in](#) by fornode

[lkjkj](#) by John87

[o](#) by John87

[.](#) by John87

[ncaa replica jersey cheap](#) by John87

[Designer Handbag Knockoffs](#) by John87

[Thanks for this great site!](#) by John87

[About TCSC](#) [Annual Reports](#) [Conferences](#) [Technical Areas](#) [Young Researcher Forum](#) [Join TCSC](#)
[Awards](#) [Book Donation](#) [Newsletters](#) [Press Articles](#) [Regional Forums](#) [TCSC Mailing List](#)
[Discussion Forums](#)

© IEEE Technical Committee on Scalable Computing