Manual Monitor

Overview

The monitor, a suitable name is not yet invented, is a application life cycle monitor, with emphasize on performance metrics.

This means that the monitor is used to collect metrics, for longer periods in production like environments, with very low intrusive overhead. And that the metrics are to be used to analyze performance issues.

The monitor consists of a central server process, acting as a database, and a set of agents that collect metrics from the environment. Users can connect to the server process and retrieve graphs of stored metrics, correlate with other metrics and other periods of time.

This is still a work in progress, and a stable release has not been released. This does also utilize a berkeley database that is tied to the SleepyCat Oracle license. This means that you can't use or embed this as a commercial product. You are free to analyze commercial products.

The monitor is mainly developed in Clojure, some Java and some C#.

Installation and Configuration

Monitor

The Monitor software is self contained and consists of the monitor.jar, a configuration file, and windows executable. An example configuration conf.clj is provided. The monitor consists of a central process for the monitoring, acting as a database. It utilizes a number of agents to collect metrics from various locations, like Java processes, Linux kernel and Windows operating system.

Monitor require at least Java SE 6, which can be downloaded from:

http://java.sun.com/javase/downloads/widget/jdk6.jsp

Common for all Monitor software except the client user interface, and the Telnet Perfmon Service is that each by default create a small control user interface, from which you can stop the process gracefully. This user interface can be eliminated by setting java.awt.headless property to true. In headless mode you can terminate gracefully using JMX. Termination is also possible by creating specially named files in the user home directory.

Monitor server

The monitor server acts as a database which store metrics for a number of days. It acts as the central for all agents running within the environment, collecting metrics from each one of them. It acts as a server for users and their user interfaces. The monitor server is configured using a configuration file, which is a Clojure program. The configuration file executes a set of agent connection definitions prior to calling the server. You configure the server by altering the agent connections you need. The following snippet is a minimal example configuration..

```
(ns conf (:use monitor.monitor))
(remote-jmx localhost 4041)
(linux-proc localhost 4042)
(in-env 100 "db" 3030)
```

The first line is required and instructs the Clojure program that it is named **conf**, and that it uses functions defined in the name space **monitor.monitor**. The latter is the interface towards the monitor server. Functions used in the configuration are defined in name space **monitor.monitor**.

The second line instructs the monitor server to collect metrics from a jmx agent found on localhost, that are listening on TCP port 4041.

The third line instructs the monitor server to collect metrics from a linux kernel agent, on **localhost**, listening on TCP port **4042**.

The last line instructs the monitor server to execute in an environment where data is kept for **100** days, in a database called "**db**", and that user interface clients are welcome to connect to TCP port **3030**.

The configuration file is a simple clojure program, and clojure programmers are free to create more advanced configurations, as the configuration simply is a clojure program. The function **in-env** won't return until the monitor server is terminated.

The function **in-env** has an additional and optional integer parameter which define the RMI port used by the client. This additional parameter is useful when the monitor server is behind a firewall that has been configured with opened ports.

The database name "**db**" is a directory in current working directory, that will be used to store data. This directory is exclusive for each monitor server. Concurrent monitor servers have to use different directories.

The monitor server can be gracefully shutdown by creating the file **~/.shutdownmonitor**

Monitoring JMX

Java processes are monitored using JMX. The agent is contained in the monitor jar, and is started with jmx as the first java command line argument. The rest of the arguments should be listener port, on which the agent should listen for the monitor process, followed by pairs of application names and their corresponding JMX port. An application name is a constant string that the user uses to identify a process, independent of how many times it has been executed. If your system consists of a web server and a back end data store, suitable application names ought to be web and datastore. Alternatively, if there is only additional argument to jmx, that argument should correspond to a file, containing the rest of the command line arguments. Any line beginning with a semi colon, **;**, in that file is considered a comment.

Example of a jmx agent command line, which monitor JMX metrics of processes exposing their platform MbeanServers to port **3901** and **3902**. We call these processes web and datastore.

```
java -jar Monitor.jar jmx 4041 web 3901 datastore 3902
```

The following example of a jmx agent command line, which monitor JMX metrics defined within a file.

```
java -jar Monitor.jar jmx configFile
```

The file **configFile** ought to include the following lines, if the same processes as the example above is to be monitored.

```
4041
web 3901
datastore 3902
```

In the server configuration file, you define the connection to the jmx agent using the **remote-jmx** funktion call, wich takes the arguments host name and port. If you e.g. have a agent running on your **localhost** on port **4041**, you need a line like (`jmx-remote "localhost" 4041)` in your configuration file.

Java processes may be monitored, using JMX, from the monitor server directly, without a JMX agent. This will not work sufficient through firewalls, since JMX uses RMI over random TCP ports. Using the monitor server directly is however easier when firewalls aren't an issue. Direct JMX is configured using the **java6-jmx** function call, witch takes the three arguments name, host and port. A call in the configuration file might look like `(java6-jmx web localhost 3901)`

The jmx agent can be gracefully shutdown by creating the file **~/.shutdownjmxprobe**

Monitor Linux kernel

The monitor contain a linux kernel agent, which scans the linux proc file system for metrics. The Linux agent is started when the first java command line argument is linux. The other command line arguments are TCP port and a regular expression, defining matches of command lines of those processes you need additional metrics from. An example would be:

```
java -jar Monitor.jar linux 4042 java
```

This command line would create a linux kernel agent that listens on port **4042**, and fetches additional metrics from processes which contain the string "**java**" in their command lines. That is the linux agent will emit process metrics for all java processes, since these have the text java in their command line.

The linux kernel agent can be gracefully shutdown by creating the file **~/.shutdownlinuxprobe**

Monitor user interface

The user interface is also contained in the Monitor.jar file, and is started without any java command line arguments. The user interface is a Swing application, that communicate with the Monitor server using RMI. Alternativly you can start the user interface by applying the arguments **gui** followed by the host and port that direct you to a particular server.

Its possible to prevent the JVM from terminating when the user interface is closed, by setting the system property **stayalive** to true.

The Monitor.jar file does not have to be the very same instance as the server process. You can distribute a copy of the jar file to any client machine. It does not require the configuration file either.

Telnet perfmon service

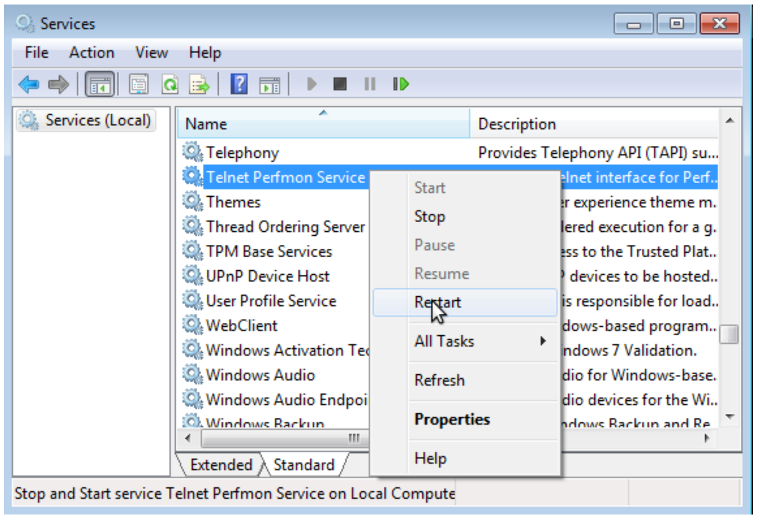
Telnet perfmon service is a small windows service, that collects perfmon metrics from windows hosts.

The Telnet perfmon service require .NET 3.5 Runtime, which can be downloaded from:

http://www.microsoft.com/downloads/details.aspx?FamilyID=333325fd-ae52-4e35-b531-508d977d32a6

To install the Telner perfmon service you simply copy the binary PerfmonMonitor.exe to a suitable location, where it will live, and execute it. The process will install itself as a service and start to execute immediately. You unintall the Telnet perfmon service by executing it with -u as a command line switch. The Telnet perfmon service require administrator privileges.

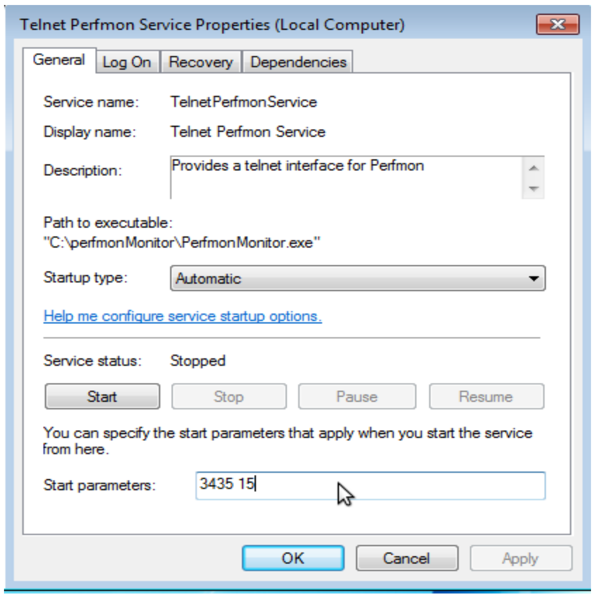
The Telnet perfmon service does not collect all perfmon metrics found on the computer. As that would be very intrusive. Up on first start, it will create a **categories** file in the same directory as where the binary resides. That is in the executing assembly location. This file contain a list of categories that the Telnet perfmon service will collect. You may alter this file, but it requires a restart of the service. Restart is easiest done using the service program.



Name	Description
Telephony	Provides Telephony API (TAPI) su...
Telnet Perfmon Service	elnet interface for Perf...
Themes	r experience theme m.
Thread Ordering Server	lered execution for a g.
TPM Base Services	ss to the Trusted Plat..
UPnP Device Host	devices to be hosted..
User Profile Service	is responsible for load..
WebClient	dows-based program..
Windows Activation Tec	ndows 7 Validation.
Windows Audio	dio for Windows-base.
Windows Audio Endpoi	dio devices for the Wi..
Windows Backup	ndows Backup and Re

Stop and Start service Telnet Perfmon Service on Local Compute

By default the Telnet perfmon service is listening on port **3434**. This can be altered by altering the command line. The first parameter is the listener port, while the second parameter is the sampling interval. This defaults to every **15** seconds.


Telnet Perfmon Service Properties (Local Computer)

Service name: TelnetPerfmonService

Display name: Telnet Perfmon Service

Description: Provides a telnet interface for Perfmon

Path to executable:
"C:\perfmonMonitor\PerfmonMonitor.exe"

Startup type: Automatic

Help me configure service startup options.

Service status: Stopped

You can specify the start parameters that apply when you start the service from here.

Start parameters: 3435 15

The following function is an example configuration for gathering metrics from a windows host.

```
(perfmon “winmachine” 3434
 “Processor|System”
 "User Time|Processor Time|Interrupts/sec|Context Switches/sec”)
```

This function call instructs the monitor to collect metrics from the **Processor** and **System** categories on host “**winmachine**”. The counters **User Time**, **Processor Time** and **Interrupts/sec** is collected from **Processor** category, while **Context Switches/sec** is collected from **System** category. The Telnet perfmon service is listening on default TCP port **3434**.

The perfmon call does also contain an additional instance parameters. In order to only collect the total metrics from the Processor category, the call would look like:

```
(perfmon “winmachine” 3434
 “Processor|System”
 "User Time|Processor Time|Interrupts/sec|Context Switches/sec”
 “_Total”)
```

The three parameters corresponding **Category**, **Counter** and **Instance** are all regular expressions.

Usage

Monitor

The monitor binary consists of both the client and the server programs. The client is used to connect to the server, which is collecting and storing metrics from defined processes. The client visualizes the data according to the user. The client is started by simply clicking on the monitor.jar file from within the explorer or by running the command:

```
java -jar monitor.jar
```

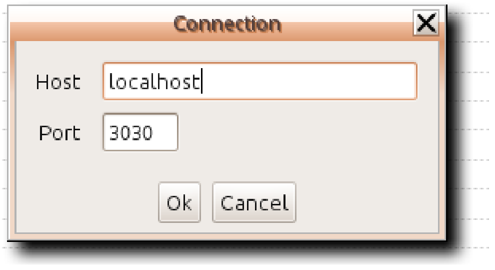
Server

The server is started by adding "**server**" to the command line, followed by the name of the configuration script. The configuration script is assumed to be located in the same directory as the monitor.jar file.
The configuration script is a clojure program, which starts the server with a set of functions supplied.

User interface

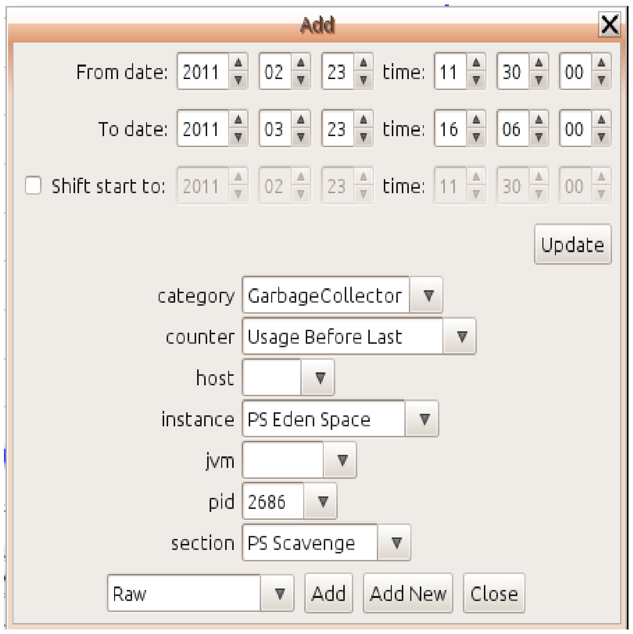
The user interface is started by running **java -jar monitor**, or by simply clicking on the **monitor.jar** in the file explorer. At start up a analysis window is opened.

The analysis window consists of an empty graph and a empty table, since no data has yet been selected. To connect to the server, you simply chose Connect from the File menu, which brings up the following form, in which you enter the address to the monitor server of interest.



Host: localhost
Port: 3030

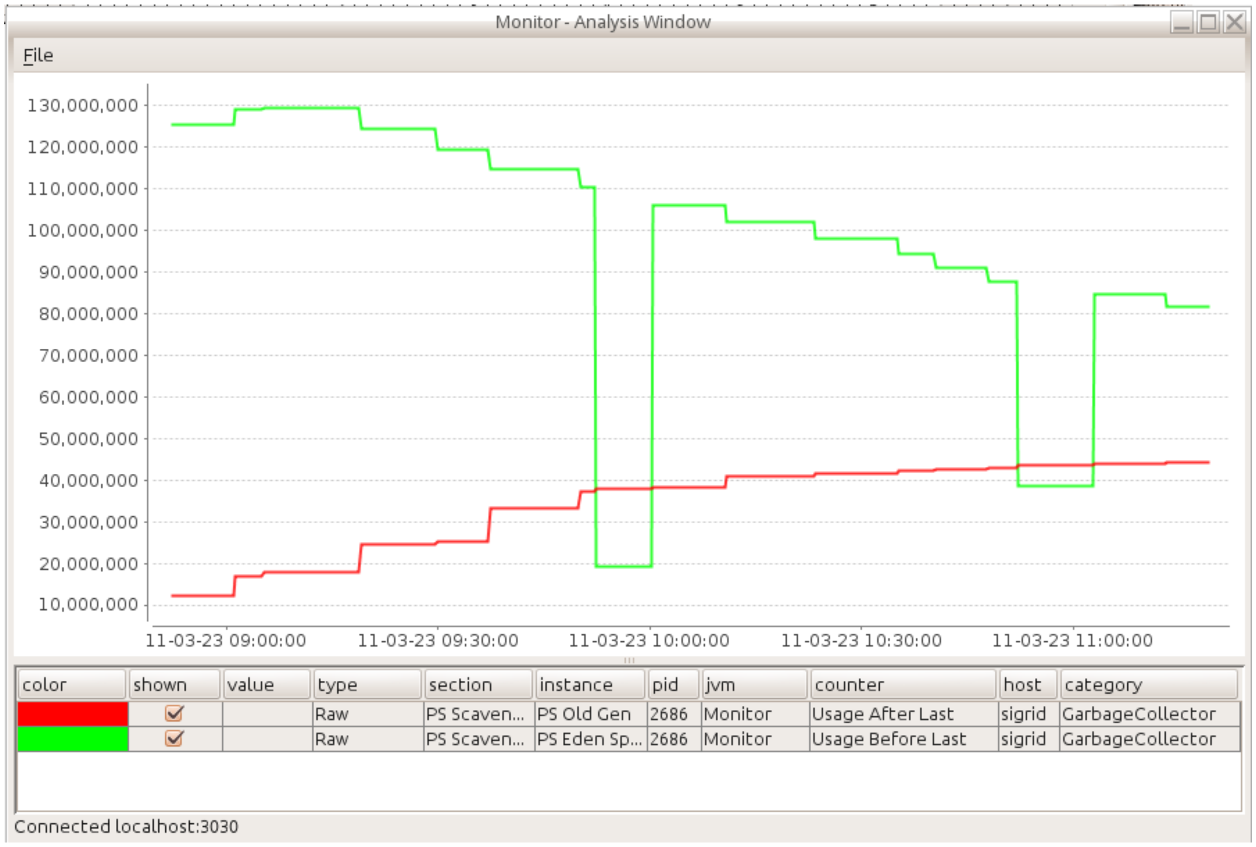
Once connected, you select the fill the graph by selecting Add in the File menu, which brings up a form where you can select data to analyze. First you select the time span containing data of interest. After update has been pressed, the available selection criteria for this period shows up. Below is an example where Garbage collection statistics of a process using the Parallel Scavenge Collector is being selected.



From date: 2011 02 23 time: 11 30 00
To date: 2011 03 23 time: 16 06 00
Shift start to: 2011 02 23 time: 11 30 00

category: GarbageCollector
counter: Usage Before Last
host:
instance: PS Eden Space
jvm:
pid: 2686
section: PS Scavenge
Raw

Note that the criteria available is dependent of both the time span, and the criteria currently selected. Here the instance criteria is only showing values available to garbage collection. That is how much memory that was used for a certain generation directly after the last GC. After pressing the Add button. the corresponding graphs should appear.



color	shown	value	type	section	instance	pid	jvm	counter	host	category
🟥	✓		Raw	PS Scaven...	PS Old Gen	2686	Monitor	Usage After Last	sigrid	GarbageCollector
🟩	✓		Raw	PS Scaven...	PS Eden Sp...	2686	Monitor	Usage Before Last	sigrid	GarbageCollector

Connected localhost:3030

The criteria available is dependent of what the server is configured to collect, which in turn is dependent of e.g. the Telnet Perfmon Service are configured to collect. The analysis view will only show the criteria available for the selected time span, while runtime view will only show criteria available during the last hour. The Java 6 JMX related criteria differ from JVM to JVM. Different Garbage collectors does i.e. produce different statistics.

When selecting the time critera, you may also define a shift of start time. When this is selected, all values will be time shifted with difference between time shift specification and start time. This is useful when analyzing behavior over time, since you easily can compare values from different periods in the same graph.

Configuring Telnet Perfmon Service

Perfmon on windows works with categories, which in turn contain counters and instances. Each time a sample is taken from the OS, a whole category sample is taken. The categories available differ from host to host. Some categories are very large. E.g. the Threading category, since a system most often use a lot of threads.

Up on first start the telnet perfmon service creates a categories file in the same directory as where the binary reside. The categories file can be altered to collect other perfmon categories than the default ones. The service has to be restarted after modification.

The next step is to instruct the Monitor server to collect perfmon data. This is done in the server configuration file. Configuration perfmon data is done by filtering, using regular expressions. The following form instructs the monitor to collect perfmon metrics from the telnet perfmon service running on **winmachine** and is available on TCP port **3434**.This also tells that the Telnet perfmon service should filter out categories matching the **Processor|System** regular expression. Within those Categories we are interested in Counters matching the regular expression "**User Time|Processor Time|Interrupts/sec|Context Switches/sec**". Within the Processor counter, we want to collect the **Total** instances, and not each individual processor that the winmachine has, thus the "**_Total**" instance regular expression. The System category does not contain instances, and the "**_Total**" regular expression wont hence filter any counters from this category.

```
(perfmon "winmachine" 3434
 "Processor|System"
 "User Time|Processor Time|Interrupts/sec|Context Switches/sec"
 "_Total")
```

Use the perfmon windows application when figuring out which metrics to collect.

While adding more perfmon metrics to your system, you need to write more and more advanced regular expressions that filter out data of interest. The monitor server saves all data collected, and you don't want to fill your database with huge amounts of non interesting data. This can be complicated, since different categories might contain counters and instances with same or similar names. It is possible to use multiple perfmon statements to the same Telnet Perfmon Service. This consumes resources, but could still be a viable alternative, compared to the resource utilization of collecting too much.

JMX Metrics

Garbage Collection

The following Garbage Collection metrics are commonly found by the JMX agent. These are found by selecting category **Garbage Collection**. Java usually utilize generational heaps, managed concurrently by different collectors.

In garbage collection metrics, you select heap using the **instance** field, and collector using the **section** field. Names vary with JVM, but in section there is a element called **Latest** with selects the latest statistics independently of collector.

Usually there are a number of spaces named something like young, eden and survivor that is managed by young collector. The young collector is usually named something like scavenge, new or young. Most old generation collector does collect in both young and old generations. The CMS collector does not. Two other spaces are commonly of interest Permanent space and Code Cache. Permanent Space is meant for objects that should not be collected. This space contain meta data of various kinds. It is usually heavily used and might have to be increased manually if the application generate code in runtime. Code Cache is used by the JIT compiler, and contain native code. The JIT won't be able to compile if this space becomes full.

Its important to understand that these values are snapshots, and reflect only the last GC event. Many collections might have occurred between two snapshots. Consult the **Count** metric.

Committed After Last – Is a metric that tells how much memory the JVM has allocated for a particular Heap. This metric can be displayed in absolute values or percent of maximum heap allowed. If committed is very close to maximum you may be close to running out of memory. However, a perfect system does utilize all memory, without running out of memory. Note also that the maximum heap allowed can change through time on modern JVMs. The heap size is normally altered by the garbage collector. This metric shows the value after the last garbage collection.

Max After Last – Is a metric that shows the maximum allowed heap after a garbage collection. This value may change over time on modern JVMs..

Usage – Is a metric that shows the current heap in use. That is, heap that is filled with data. This metric can be shown as usage immediately before the last garbage collection, immediately after the last garbage collection, or percent of committed heap after the last garbage collection. The two latter is the most interesting metrics in order to understand how much memory the JVM is utilizing. That is, how much data there are after garbage has been reclaimed. Looking at usage before and after on both young and old generation might give you a indication of your propagation rate. A high propagation rate, and frequent collection in old space is an indication that objects get propagated prematurely.

Count – Is a metric that tells how many collections a particular collector has performed.

Last Duration and Time – Are metrics that tell how long time the last collection took, and accumulated collection time. Low number are desirable, since most collectors will freeze your application threads during collection.

Memory

The memory counters reflect snapshots of memory regions.

Heap used – These metrics show the current over all heap utilization. That is the amount of memory occupied by objects. This metric can be shown as absolute value or in percent of maximum allowed.

Heap committed - These metrics reflect snapshots of heap size. That is, heap memory allocated by the JVM. This metric can be shown as absolute value or in percent of maximum allowed.

Non heap – These metrics reflect snapshots of non heap memory usage. It's not very well specified what this memory is used for. It's internal for the JVM.

Usage – These metrics reflect snapshots of heap utilization of each heap space.

Linux Metrics

The linux probe collects different metrics dependent of kernel version. Some important are:

CPU

user, **system**, **idle**, **iowait**, **irq**, **softirq** – These metrics tell the cpu utilization of the OS. User is time spent in user mode. That is time spent by applications. System is kernel time. High value indicate Locking, or IO issues. Idle, time spent doing nothing. Iowait is time spent waiting on IO. Irq, and softirq is time spent handling interrupts.

The above values are percent of total, that is the value 100 mean that all CPU are fully occupied.

Context switches/s and **interrupts/s** is metrics telling the average number of context switches and interrupts made per second. Context switches is the rate in witch threads get attention by the scheduler. There are many reasons for this. Memory barriers, IO events, yielding and others. Interrupts is the rate in with interrupts are handled. If the function of context switches and interrupts starts to deviate, you are experiencing exhaustion. Often lock contention.

If the linux agent is monitoring processes, the **system**, **total**, and **user** metrics will be collected. System is time spent in kernel and while user is time spent in user, and total is all time spent in the process.

Memory

The linux agent will likely provide the following metrics in the memory category:

physical used, physical free and % available – Amount of RAM currently used, and amount of RAM currently free. Note that Linux usually spend most of the physical memory available for disk cache. That is to reduce I/O. That means that physical free usually is a lot less than what actually is available for use. The % available reflect percent of memory available for allocation, including memory currently used as cache.

Counters **cache and % cache**– Cache reflects the amount of memory that is used as disk cache, and % cache reflect the cache percentage of all physical memory. Cache include cache of swap and buffers.

swap used, **% swap**, **% swaped** – Reflect amount of swap used, percent of swap used, and percent of all memory dedicated to swap. The latter include the whole RAM and swap space.

Per process counters are:

resident – Amount of RAM used by this process. Includes both shared and private pages.

swapped – Amount of memory swapped out for this process. Includes both shared and private pages.

private – Amount of memory exclusively used by this process. Includes both resident and swapped pages.

shared – Amount of shared memory used by this process. Includes both resident and swapped pages.

proportional – Amount of memory proportionally used by this process. The size of each shared page is divided the number of processes sharing it.

Disk

Written kB/s , **read kB/s** – Reflects the average number of kilo bytes written and read from disk per second.

Writes/s , **reads/s** – Reflects the physical number of write and read operations performed per second. A merged write is viewed as a single write. It is effective to transfer a lot of data in few operations.

Queue – Reflects the number of queued disk operations.

Network device

Received kB/s , **sent kB/s** – Reflects the number of kilo bytes received and sent on the network device.

Received packets/s , **Sent packets/s** – Reflects the number of packets sent or received on the network device. It is effective to transfer a lot of data in few packets.

Descriptors

All and **% All**– The number of file descriptors each monitored process is having opened, and % of maximum. That is the soft file descriptor limit.

Thread monitoring on Java

Its possible to perform basic thread monitoring of Java processes through JMX. Many server side Java processes contain a huge number of threads, so to prevent intrusiveness, threads aren't monitored using the JMX agent, since the agent is meant to run continuously. There is however a thread monitoring tool provided in monitor, that may be used to monitor threads temporarily.

Thread monitoring can either be done by a direct connection to the server, or by creating a file, that is later imported into the server. The latter is useful if the monitored process resides behind a firewall or if you don't want to be intrusive on the network while the threads monitoring is in progress.

To monitor with a direct connection to the server you issue the command

```
java -jar monitor.jar threads <name> <jmx-host> <jmx-port> <server-host>
<server-port>
```

where **name** is the name of the java application, **jmx host** and **port** is the connection to the jmx agent running within the java process, and **server host** and **port** is the connection used to connect to the server. The later is the very same parameters used to connect to the monitor server using the user interface.

The default metrics you ought get is:

CPU Time - The CPU time spent by the thread.

CPU Time User – The CPU time spent by the thread in User mode.

Blocked Count – The number of times the thread has been blocked on a Object monitor.

Waited Count – The number of time the thread has been waiting on a Object monitor.

You may instruct the jvm to provide the additional metrics:

Blocked Time – The accumulated time the thread has been blocked on a Object monitor. That is, the time the Thread has been blocked on a synchronized statement.

Waited Time – The accumulated time the thread has been waiting on a monitor. That is, the time the Thread has been waiting on Object.wait.

These counters are enabled by setting the a jmx attribute ThreadContentionMonitoringEnabled in the Threading MBean to true. Gathering data for these metrics is intrusive on the jvm, and is hence usually disabled by default. You can use e.g. JConsole to turn on thread contention monitoring.

To collect Java Thread metrics into a file use the following command:

```
java -jar Monitor.jar threadsfile <name> <jmx-host> <jmx-port> <file>
<seconds-per-sample>
```

where **file** is the path to the file that the metrics is written to, and **seconds-per-sample** is the sample interval in seconds.

To import a file with thread metrics, you use the following command

```
java -jar monitor.jar importthreadsfile <file> <server-host> <server-port>
```