PCP: Ingest and Export

pcp-conf2018 Mark Goodwin

mgoodwin@redhat.com

@goodwinos

PCP Ingest / Export

Ingest

Standard Agents

Specialized agents:
- MMV
- BCC
- Trace
- Prometheus
- .. many others

LOGIMPORT(3)
Ingest tools: xxx2pcp

PMCD
libpcp

Archive
Logs

Export

CLI tools, scripts

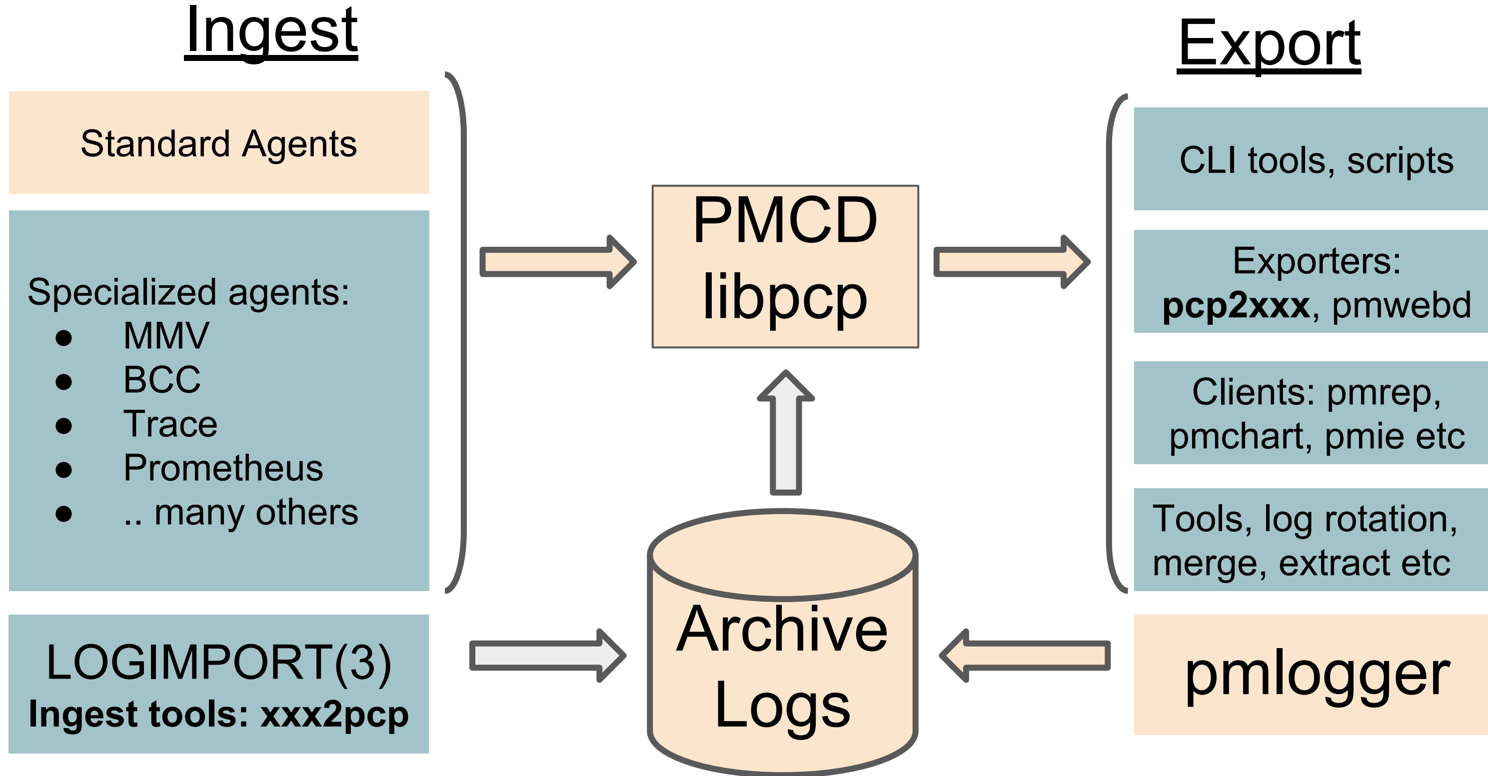
Exporters:
pcp2xxx, pmwebd

Clients: pmrep,
pmchart, pmie etc

Tools, log rotation,
merge, extract etc

pmlogger

PCP Ingest: Standard PMDAs

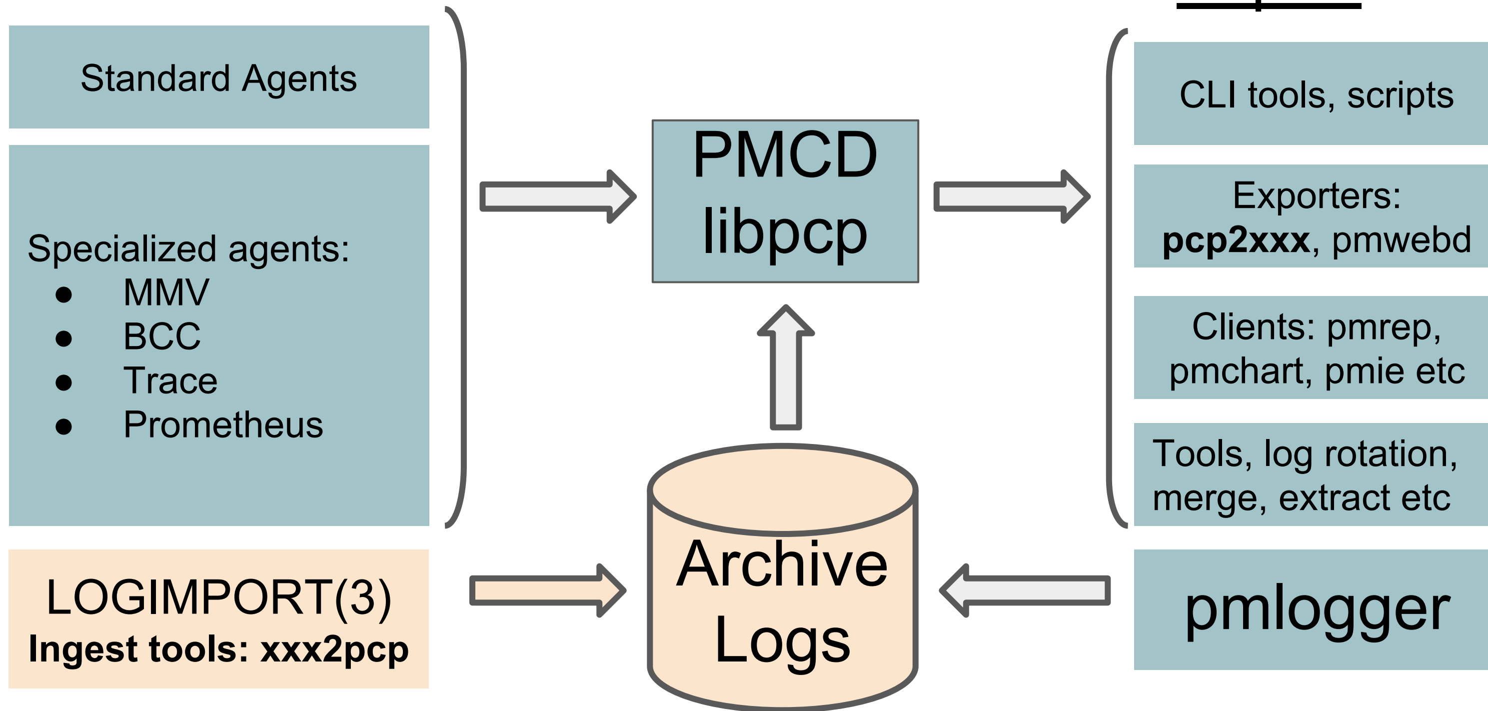


PCP Standard PMDAs (Agents)

- ~ 75 plugins / agents (PMDAs)
  - .. more being added every release
  - Managed by the PCP pmcd service.
  - DSOs and daemons. Lots of IPC options
- Ingest data into PCP metrics
  - Canonical, uniform name space
  - strongly typed metadata and values
  - Low overheads: “Pull” model: service to completion: client request -> pmcd -> agent -> pmcd -> client
- Extensible API
  - libpcp_pmda has C/C++, Python and Perl bindings
- Separately Packaged: pcp-pmda-*foo*
  - Isolate exotic dependencies
  - Not all installed by default.

- **linux** - kernel metrics. CPU, Disk, Network, Memory, Filesystem, etc. everything exported by /proc, /sys and most other kernel interfaces
- **proc** - per-process metrics
- **XFS** - XFS filesystem specific metrics
- **nfsclient** - NFS client stats
- **mmv** - memory mapped instrumentation
- **dm** - device mapper and LVM
- **jbd2** - journal block device
- **lio** - Linux I/O - iSCSI, FCP, FCoE
- **pmcd** - PCP statistics
- **root** - container, privileged PMDAs, etc
- **apache** - web server stats
- **BCC** - Extended Berkley Packet Filter metrics
- **docker** - container management stats
- **KVM** - libvirt
- **mysql** and **postgresql** - database stats
- **prometheus** - end-points
- **redis** - system stats for redis daemons
- **samba** - filesystem
- **smart** - disk health
- **vmware** - platform stats
- … many more.

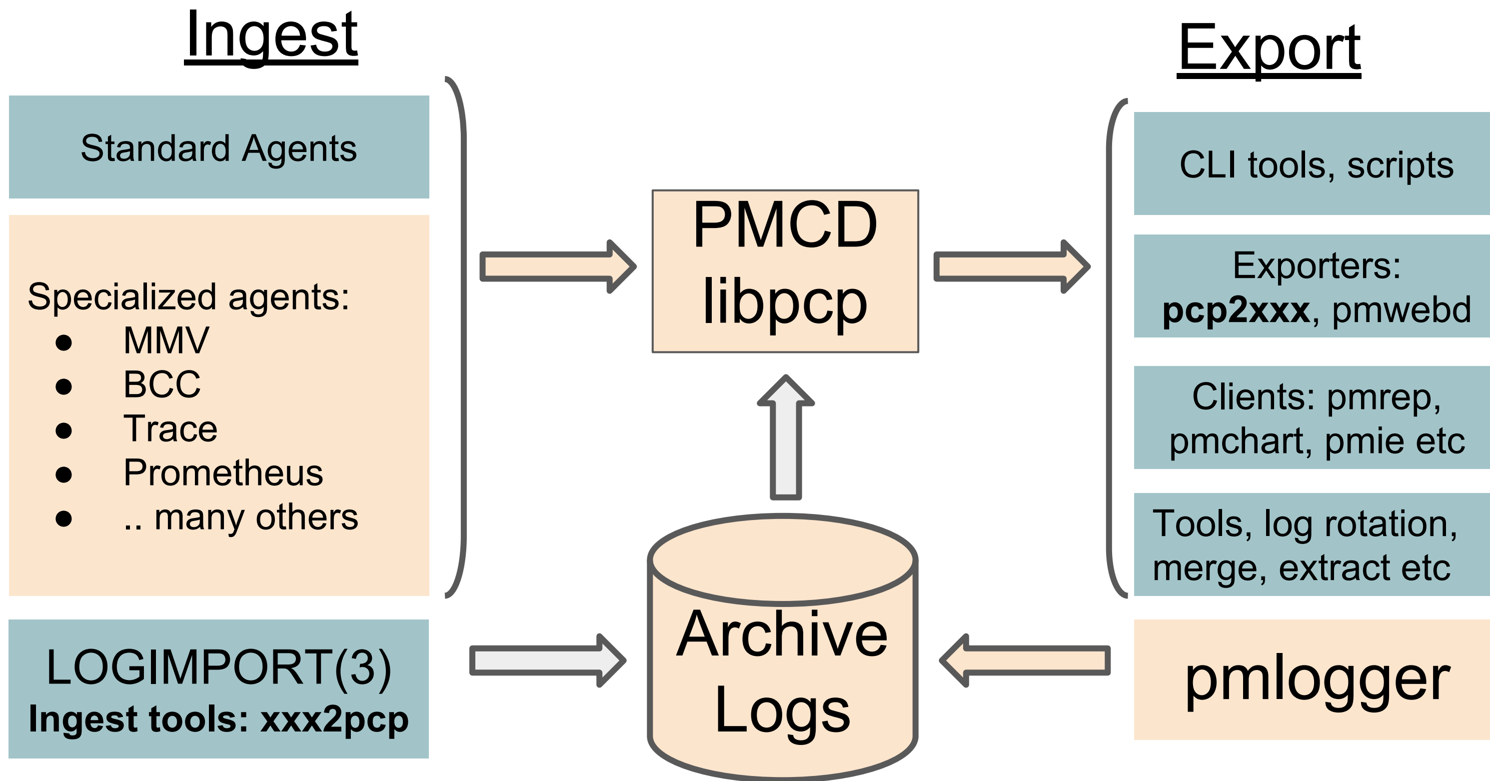
PCP Ingest: LOGIMPORT API and xxx2pcp



LOGIMPORT(3) - library to write PCP archives

- libpcp_import API for writing PCP archive logs directly
  - Provides a simple programmatic ingest interface to write PCP archives
  - By-passes normal PMDA->PMCD->pmlogger->archive data flow
- C/C++, Perl and Python bindings, with many examples
- Resulting PCP archives can be replayed/exported by any PCP tools
  - Exactly the same as standard pmlogger(1) archives
- logimport(3) is the API library behind many ***xxx*2pcp** ingest tools:
  - collectl2pcp(1)
  - ganglia2pcp(1)
  - iostat2pcp(1)
  - sar2pcp(1)
  - sheet2pcp(1)
  - mrtg2pcp(1)
  - pmrep(1) to write PCP archives, e.g. **pmrep -o archive -F *outputarchive***

PCP Ingest “ Specialized PMDAs



Specialized agents: instrumentation and tracing

- **mmv PMDA** - memory mapped values PMDA and API
  - Simple API documented in pmdammv(1), mmv(5) and mmv_stats_init(3)
  - Application and pmdammv use a shm segment
    - Suitable for very low latency instrumentation
  - Creates dynamic metrics
- **trace PMDA** - event counting / tracing PMDA and API
  - Multiple language bindings (even Fortran!)
  - pmtrace(1) can be used to instrument scripts
  - Use pmdatrace(3) API to instrument applications
  - Fixed namespace. trace.* metrics. Metric instances are trace points
- **BCC PMDA** - Extended BPF (Berkeley Packet Filter) PMDA
  - See pmdabcc(1) - stats from compiled eBPF programs loaded as kernel modules
  - Very efficient, secure and powerful, e.g. disk device i/o latency histograms
  - Extensible via ini format config file - best way to monitor kernel trace points, etc
  - Creates dynamic metrics
  - Requires pcp-4.1.0 and fairly new kernel. See bpf(2),

Specialized agents (cont.): PMDA Prometheus

- PCP PMDA to ingest prometheus end-point data
  - See pmdaprometheus(1)
  - Dynamically extensible via config files in /var/log/pcp/pmdas/prometheus/config.d
    - Each config file either contains a URL, ***or*** is an executable script.
      - Both URLs and scripts should return prometheus formatted metric data
      - file:// URLs are supported for ingesting local files.
    - Prometheus end-point metric data is simple text strings, documented at https://prometheus.io/docs/instrumenting/exposition_formats
    - Simple example:

```
# HELP mymetric Simple gauge metric with three instances
# Type mymetric gauge
mymetric {abc="0"} 456
mymetric {def="123"} 123
mymetric {hig="246",xyz="something"} 128
```

- PCP metric naming
  - The base name of each config file name is used as the second level of the resulting PCP metric names, e.g. a config file named **myserver.url** results in metrics below **prometheus.myserver** in the PCP name space.
  - Subdirectories in the config directory result in additional non-leaf namespace levels

PMDA Prometheus (cont.): **meta-data**

- PCP has strongly typed metrics and meta-data
  - Prometheus formatted metrics have no formal metadata
    - rely on loosely defined metric name hints and suffixes and the like
    - E.g. a prometheus metric name may have “_count” as a suffix to indicate it’s a counter.
  - All PCP metrics are strongly typed and have metadata
    - metric type, semantics, units and help text, see PMLOOKUPDESC(3)
  - The PCP Prometheus PMDA uses heuristics and tags to fill this in, e.g.

```
# HELP loadavg local load average
# Type loadavg gauge
loadavg {interval="1-minute"} 0.12
loadavg {interval="5-minute"} 0.27
loadavg {interval="15-minute"} 0.54
```

- Labels in the prometheus metric (e.g. “`interval`”) are used as the instance name in the resulting PCP metric data. E.g. the PCP metric **prometheus.myhost.loadavg** would have three instances.

PMDA Prometheus (cont.): **scripted configs**

- Scripted configs
  - provide a simple yet powerful way to ingest metric data. E.g. given **/proc/loadavg**

```
$ cat /proc/loadavg
0.18 0.31 0.40 1/1253 17801
```

- As an example, create an executable script in a file named **`/var/lib/pcp/pmdas/prometheus/config.d/myserver`**

```
#! /bin/sh
        awk '{
            print("# HELP loadavg local load average")
            print("# Type loadavg gauge")
            printf("loadavg {interval=\"1-minute\"} %.2f\n", $1)
            printf("loadavg {interval=\"5-minute\"} %.2f\n", $2)
            printf("loadavg {interval=\"15-minute\"} %.2f\n", $3)
        }' /proc/loadavg
```

- Results in a PCP metric named **`prometheus.myserver.loadavg`** with three instances.
  - This is created dynamically - no restarts necessary

PMDA Prometheus (cont.): **URL configs**

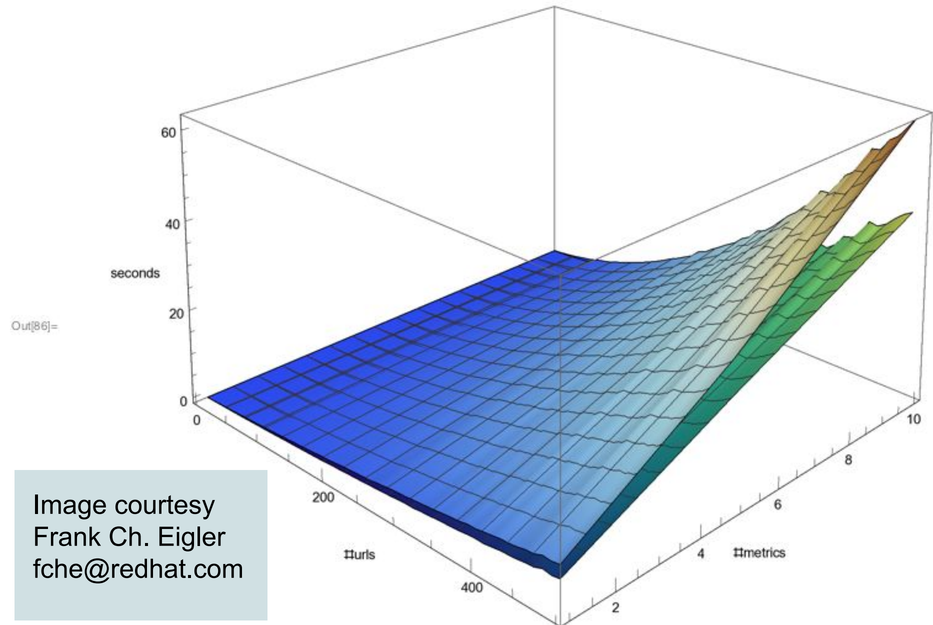
- Prometheus end-point URLs
  - Perf data exported as a URL on a port in the range 9100 - 10000 below **/metrics** e.g. http://somehost:9100/metrics is the prometheus "node exporter" for a host named *somehost*.
  - There are a huge number of prometheus exporters, PCP can ingest them all
    - See https://github.com/prometheus/prometheus/wiki/Default-port-allocations
- URL config files
  - First line in the config file is an end-point URL (as above), with **.url** suffix
  - PCP metrics are dynamically created - metrics are named same way as scripted configs.
  - No need for PMDA restarts or anything - completely dynamic
  - URL configs also support **HEADER** and **FILTER** syntax in subsequent lines in the config file
    - **HEADER** lines specify http request headers to include in the GET request
      - E.g. for authentication, content-type, proxy redirects, etc.
    - **FILTER** lines allow metrics and/or labels in the response be included/excluded
      - E.g. to exclude unwanted prometheus labels from the PCP instance domains
      - E.g. Ignore uninteresting metrics in the response, etc

PMDA Prometheus (cont).): Scalability

- Simple benchmark measuring wall clock fetch times
  - 1 to 500 URLs, with 1 to 10 metrics per URL
  - localhost http requests returning constant data (script generated)
- Scalability
  - Fairly linear scalability for #URLs with only a few metrics/URL
  - Non-linear for higher #metrics
- Mostly resolved by using parallel threads for HTTP GET requests, but serialized response parsing (FIFO queue) - avoids the “Big Python Interpreter Lock”

```
In[14]:= data := SemanticImport["/home/fche/BENCH.out_500urls_10metrics.txt"]

In[86]:= Show[
  {ListPlot3D[{data[ ;; , {1, 2, 3}]}, AxesLabel → {"#urls", "#metrics", "seconds"},
    PlotLegends → {"firstfetch"}, ColorFunction -> "LightTemperatureMap"],
   ListPlot3D[{data[ ;; , {1, 2, 4}]}, AxesLabel → {"#urls", "#metrics", "seconds"},
    PlotLegends → {"secondfetch"}, ColorFunction → "BlueGreenYellow"]}]
```



Future: