Distributed Common API for Measuring Performance

Alexander P. Sideropoulos

California Polytechnic State University, San Luis Obispo

alexander@thequery.net

October 10, 2014

Outline

Building Distributed Systems

Due to the **plateauing speed** of individual processing units and encouraged by the **interconnectedness** of the internet at large, there exists a natural trend of distributing large, complex systems across multiple components locally and throughout the world.

Notes:

1. cpus not getting faster, but core counts increasing
2. internet has become more pervasive in today's business economy
3. software systems are becoming increasingly distributed
4. EX: large user base — google, facebook
5. EX: bigger data sets — ILM, splunk
6. EX: efficient use of resources — folding@home

Building Distributed Systems

In order to effectively build these systems, software practitioners must be able to test their system for **performance defects** as well as bottlenecks, all while the distributed system itself is responding to changes in availability and work load on its individual nodes.

Notes:

1. systems are not always homogeneous w.r.t. hardware architecture or os
2. development is difficult, even with best tools

Distributed Performance Frameworks (DPF)

Distributed **Performance Testing** Frameworks:

- tools to evaluate performance of system
- both black box and white box
- instrumenting, collecting, analyzing, and visualizing distributed performance data

Distributed **Performance Monitoring** Frameworks:

- considered part of the testing framework
- black box interface
- monitoring a distributed system or application
- mechanisms for triggering actions based on performance events

Notes:

1. DPF refers to both

Distributed Performance Frameworks (DPF)

Three types of metric gathering techniques:

1. hardware counters and sensors use specialized hardware to gather highly accurate metrics and are highly dependent on the underlying hardware architecture,
2. software sensors use modern operating system interfaces to acquire moderately accurate performance metrics in an architecture-independent interface, and
3. hybrid approaches use a combination of hardware and software sensors to attain a balance between the two.

Notes:

1. frameworks listed in this work were chosen based on their categorization in the Zanikolas [3] taxonomy; only level 2 frameworks are included.
2. Level 2 frameworks are defined as having at least one type of republisher in addition to producers;
3. these frameworks usually distribute functionality across multiple hosts.
4. A limited analysis is conducted by reviewing the available literature, and further analysis (i.e., verifying scalability, transparency, and validity) is left as future work.

Criterion for Evaluating Distributed Performance Frameworks

In order for practitioners and researchers alike to **effectively choose** a distributed performance framework, it is necessary to have a set criteria for evaluation. Presented here is an extended criterion of the general requirements presented by Zanikolas et al. for grid systems [3].

1. Data Delivery Models (original)
2. Security (original)
3. Scalability (only consider good performance)
4. Transparency (replaces Low Intrusiveness)
5. Extensibility (removed)
6. Completeness (new)
7. Validity (new)
8. Portability (alternate)

Notes:

1. there exist several DPF developed by commercial and research institutions
2. **none found meets all of these criteria**
3. delivery model: periodic vs on-demand (dynamic vs static data), push vs pull (sparse vs stream)
4. security: access control, single or mutual authentication, secure transport of monitoring information
5. transparency: “typically measured as a function of host (processor, memory, I/O) and network load (bandwidth) generated by the collection, processing and distribution of events”
6. portability: run on heterogeneous systems without special considerations; black box

Distributed Common API for Measuring Performance

dCAMP is a distributed performance framework built on top of Mark Gabel and Michael Haungs' 2007 research on *CAMP: a common API for measuring performance*[1]. *CAMP* provides an **accurate** and "**consistent** method for retrieving system performance data from **multiple platforms**."

dCAMP builds on this functionality and the authors' work validating *CAMP*'s accuracy by adding these core feature sets:

- stateful performance API
- distributed performance data aggregation
- performance filters and triggers
- simplistic fault tolerance

Notes:

1. Many authors have failed to address their framework's validity, transparency, and scalability explicitly, thinking the framework's architecture speaks for itself or blindly assuming it is accurate and introduces negligible load on the measured system.
2. As shown in the analysis work presented later, *dCAMP* adds these features while still maintaining minimal impact on the systems, processes, and networks being monitored.

System Architecture

dCAMP is designed as a **semi-centralized, hierarchical peer-to-peer system** utilizing the UNIX **Pipes and Filter** architectural pattern in which leaf nodes of the hierarchy collect data, filter out extraneous data, and send it up the pipe to a parent node which subsequently filters out more data and sends it up to another parent node.

Notes:

1. This architecture is efficient in that unwanted data is discarded earlier in the data path, reducing transport and processing costs.

dCAMP Roles and Services

The *dCAMP* distributed system is comprised of one or more nodes, each executing a **role**—a named grouping of a specific, known set of **services**. Each *dCAMP* service implements a specific function.

- **Node**—rudimentary *dCAMP* functionality; topology communication, heartbeat monitoring, failure recovery
- **Sensor**—local metric gathering; essentially the *dCAMP* layer on top of OS/hardware APIs (via CAMP)
- **Filter**—metric filtering; throttling and thresholding
- **Aggregation**–metric aggregation; metric collection and calculation of multiple Sensor and/or Aggregation services
- **Management**–end-user control of *dCAMP*
- **Configuration**–configuration replication; topology and configuration distribution

Notes:

1. Roles have little to no actual run-time logic but simply act as containers for the services
2. services manage ZeroMQ sockets, communicating with other services/nodes, and do the real work of *dCAMP*
3. DEFINE: zeromq is a message queueing framework, provides sockets and message patterns for distributed software

dCAMP Roles and Services

The *dCAMP* distributed system is comprised of one or more nodes, each executing a **role**—a named grouping of a specific, known set of **services**. Each *dCAMP* service implements a specific function.

A service's scope can vary depending on the node's level in the *dCAMP* topology.

- *Root*: master copy of the configuration, publishing new values as needed
- *Collector*: stores (and forwards) every update from the *Root*
- *Metric*: only stores updates relevant to the node

Notes:

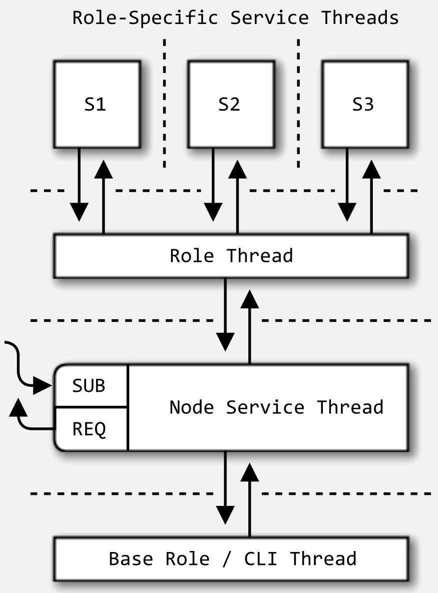
dCAMP Roles and Services

The *dCAMP* distributed system is comprised of one or more nodes, each executing a **role**—a named grouping of a specific, known set of **services**. Each *dCAMP* service implements a specific function.

Role	Service(s)
Root	Management, Aggregation, Filter, Configuration (Full)
Collector	Aggregation, Filter, Configuration (Full)
Metric	Sensor, Filter, Configuration (Partial)
Base	Node

Notes:

1. *Metric* role runs on the nodes from which performance metrics should be collected.
2. *Collector* role acts as an aggregation point in the system, from multiple *Metric* (and *Collector*) nodes; provides additional aggregated performance metrics.
3. only one *Root* role; acts as master copy of the configuration and sole user-interface point.
4. *Root* role is not strictly attached to any given node in the system; any first-level *Collector*
5. may choose to split roles across multiple nodes or collapse them onto a single node.



The *Base* role must be running on each node for it to be part of the *dCAMP* distributed system. All other roles are launched from within the *Base* role.

Notes:

1. In this document, a "*Base* node" is defined as a *dCAMP* node which has not yet been configured, i.e. it has not joined a running *dCAMP* system.
2. Node, Role, Services Threading Model Diagram: Thread boundaries are represented by dashed lines. Except for the Node service's `SUB` and `REQ` sockets, all arrows represent `PAIR` socket communication.
3. *Base* node can transform into one of the three active *dCAMP* roles; this transformation is actually the *Base* role (via the Node service) launching and managing another role internally.
4. When a *Base* node, only bottom two threads (*Base* and Node) are active.
5. after assignment from the "discover" Topology Protocol or CLI, Node service launches an appropriate role thread which, in turn, launches one or more role-specific service threads.
6. also various service-to-service communications via `inproc` sockets (e.g. internal Data Flow Protocol) and shared memory (e.g. the Configuration service).

Sequence of *dCAMP* Operation

1. User promotes a *Root* node via the *dCAMP* CLI, specifying a configuration file and a *Base* node's address.
2. *Root* node connects to each *Base* node and begins the "discover" Topology Protocol.
3. *Base* nodes join the *dCAMP* system at any time, being assigned as *Collector* or *Metric* nodes in the topology.
4. *dCAMP* runs in a steady state, nodes entering or exiting the system at any time.
5. User stops *dCAMP* by using the *dCAMP* CLI command.
6. *Root* node begins the "stop" Topology Protocol.
7. *Collector* and *Metric* nodes exit the topology and revert to *Base* nodes.
8. *Root* node exits, reverting to *Base* node.

Notes:

1. *Base* nodes (other *Root*) can be started at any time by using the *dCAMP* CLI
2. expected *Base* nodes are managed by a watchdog utility

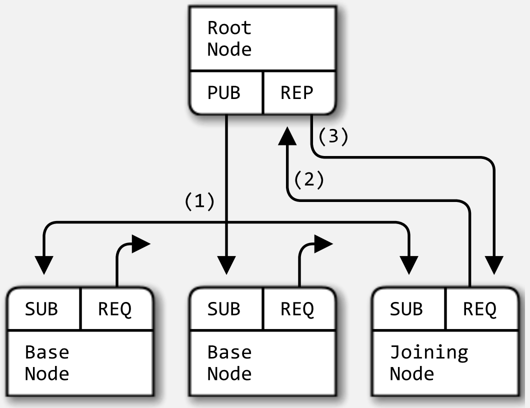
Steady-State Operation

1. Performance counters are sampled, filtered, reported, and logged by the Metric nodes at regular intervals according to the *dCAMP* Configuration.
2. Performance counters received from child nodes are aggregated, filtered, reported, and logged by *Collector* nodes at regular intervals according to the *dCAMP* Configuration.
3. Performance counters received from child nodes are aggregated and logged by *Root* node for later processing (e.g. graphing metrics during a test scenario or correlating statistics with a distributed event log).

Notes:

Topology Protocol

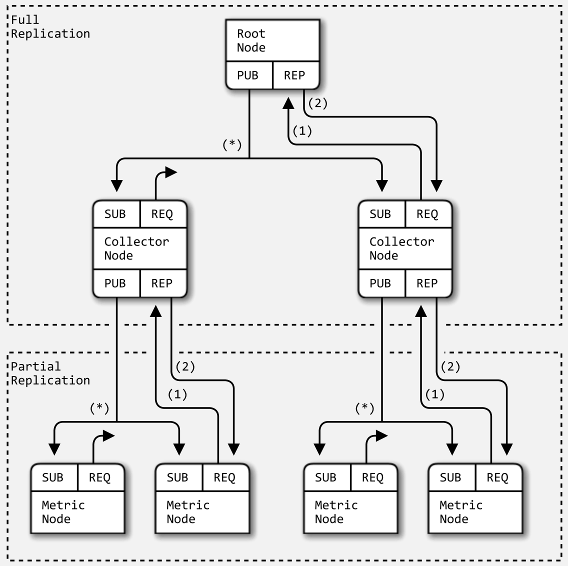
The *dCAMP* distributed topology is dynamically established as the *Root* node sends out its discovery message and receives join messages from *Base* nodes. When a *Base* node responds to the *Root*, the *Base* node is given its assignment.



Notes:

1. To reduce network traffic and load on the *Root*, *Base* nodes are designed to ignore `MARCO` messages from nodes whose UUID matches a previous successful topology discovery handshake.
2. The *Root* node uses this to its advantage when attempting to stop nodes: the same `MARCO` / `POLO` pattern is used, but the *Root* node uses a different UUID in the `MARCO` message and a responds with a `STOP` message instead of an assignment.

Configuration Protocol



Notes:

1. *dCAMP* configuration and topology state are replicated across the system using key-value pairs, with the keys laid out in a hierarchical fashion.
2. *Metric* node only needs the configuration values for its particular group, the node subscribes only to the `"/CONFIG/<group-name>/"` topic.
3. Any `KVPUB` whose key does not start with this string is then discarded.

Configuration Protocol

The *dCAMP* configuration replication algorithm generally adheres to the Clustered Hashmap Protocol[2].

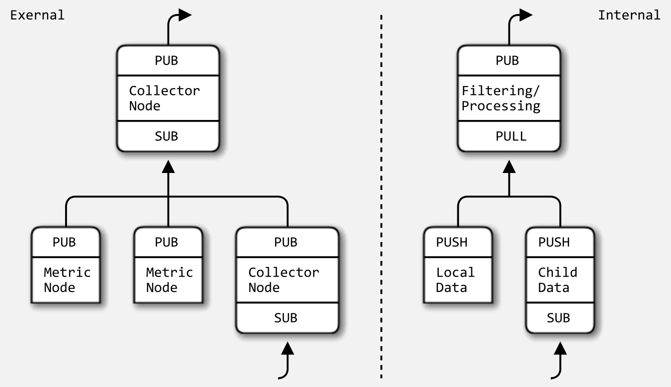
```
config-replication = *update / snap-sync
update             = P-KVPUB / P-HUGZ
snap-sync          = C-ICANHAZ ( *P-KVSYNC P-KTHXBAI )
```

Notes:

1. A newly assigned **first-level Collector** node will first subscribe to new configuration updates from the *Root* node and then send a configuration snapshot request to the *Root* node.
2. A newly assigned **Metric** (or non-first-level *Collector*) node will first subscribe to new configuration updates from its parent *Collector* node, and then send its parent *Collector* node a **filtered configuration snapshot request**.
3. Once its snapshot has been successfully received, a node will process any pending configuration updates and then, in the case of a *Collector* node, respond to child node snapshot requests.

Data Protocol

There are two data flow protocols in the *dCAMP* system: the **external protocol** for data flowing between nodes and the **internal protocol** for data flowing between components within a single node.

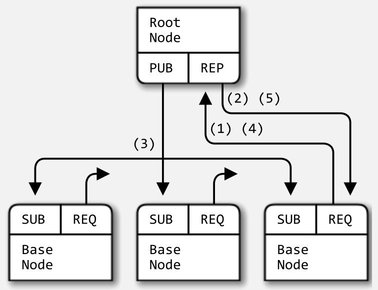


Notes:

1. Both protocols have the same specification and use the same message formats.
2. The *dCAMP* data flow protocol is very simple: single message type.
3. The data flows from one node to another via PUB/SUB sockets.
4. Internally, data flows from the upstream data producers, through a filtering/processing unit, and out to downstream data consumers via PUSH/PULL sockets.
5. When data rate is slower than a predefined threshold, heartbeats are sent instead to keep inter-node connections alive.

Recovery Protocols: Promotion

The *dCAMP* Recovery Protocols are used for the Promotion and Election algorithms and use the same base messages as the Topology Protocol.

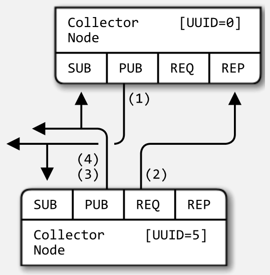


Notes:

1. The Branch Recovery Protocol is initiated by *Metric* nodes when they detect their *Collector* has died.
2. Once the *Root* node has received an `SOS` message from at least one third of the branch's *Metric* nodes, the *Root* proceeds to shutdown the entire branch using the "stop" Topology Protocol.
3. Once shut down, a new *Collector* is selected and the branch is rebuilt using the standard "discover" Topology Protocol.

Recovery Protocols: Election

The *dCAMP* Recovery Protocols are used for the Promotion and Election algorithms and use the same base messages as the Topology Protocol.



Notes:

1. As each *Collector* node detects the *Root* node has died, it attempts to start an election via the `WUTUP` message.
2. *Collector* nodes with higher UUIDs will respond to the first *Collector* by sending the `YO` message.
3. If no `YO` messages are received by the first *Collector*, the `IWIN` message is sent out to all *Collector* nodes, self-declaring the first *Collector* as the new Root.

Strategy

To measure the impact of *dCAMP* on a monitored process, a workload is defined and measured with and without *dCAMP* active. The measured difference in performance of the monitored process is defined to be *dCAMP*'s monitoring overhead.

Notes:

Workload

Client: Apache JMeter (v2.11) on a MacBook Pro (2.7GHz Core i7, 8GB 1333MHz DDR3, SSD) running OSX 10.9.
Server: default-configured Apache instance on a Lenovo Thinkpad (dual 2.16GHz Centrino Duo T2600, 2GB 667MHz DDR2, SATA) running Ubuntu 13.10.

Each test run includes 18 different load points, scaling the number of client threads from 2 to 2048. For every load point, the threads continuously (in this order):

1. load a static home page,
2. load a PHP page which calculates the 25th Fibonacci number, and
3. download a 5MB file of random binary data.

Notes:

1. The client machine is directly connected to the Apache server via crossover gigabit Ethernet.
2. Fibonacci workload is CPU-bound, 5MB download is disk-bound, home page workload is only used to seed the client connection and is not part of the analysis measurements.
3. After the ramp up phase of each load point (launching 10 threads per second), the test ensures all threads continue to execute simultaneously for five minutes before shutting down.
4. The arithmetic mean of the request latency for each step at each load point is then calculated and averaged across three distinct runs of the same test.

Configuration

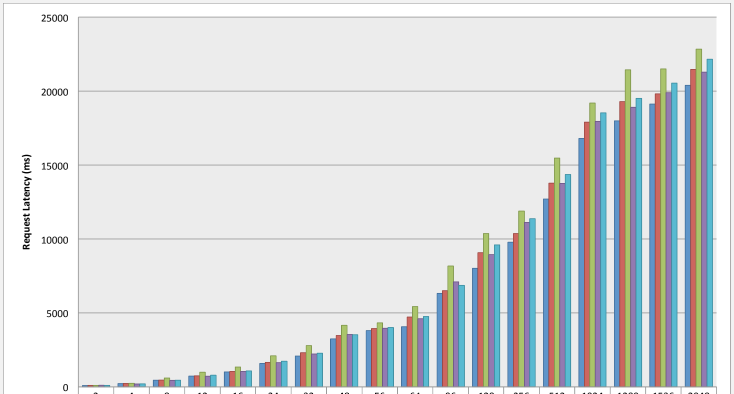
Each *dCAMP* configuration level monitors **four global metrics** and **three process-specific metrics** on the Apache process(es). The global metrics are CPU usage (`proc`), memory usage (`mem`), disk throughput (`disk`), and network throughput (`net`); the Apache metrics are CPU usage (`apache_cpu`), memory usage (`apache_mem`), and combined disk/network throughput (`apache_io`).

- **baseline** – *dCAMP* off
- **5m** – all metrics every 300 seconds, heartbeats every 60 seconds
- **1m** – all metrics every 60 seconds, heartbeats every 60 seconds
- **10s** – global metrics every 300 seconds, Apache metrics every 10 seconds, heartbeats every 300 seconds
- **1s** – global metrics every 300 seconds, Apache metrics every 1 second, heartbeats every 300 seconds

Notes:

1. No thresholds were defined for any of the configurations. That is, Sensor nodes immediately reported every sample instead of holding them for later reporting.

CPU-Bound Results

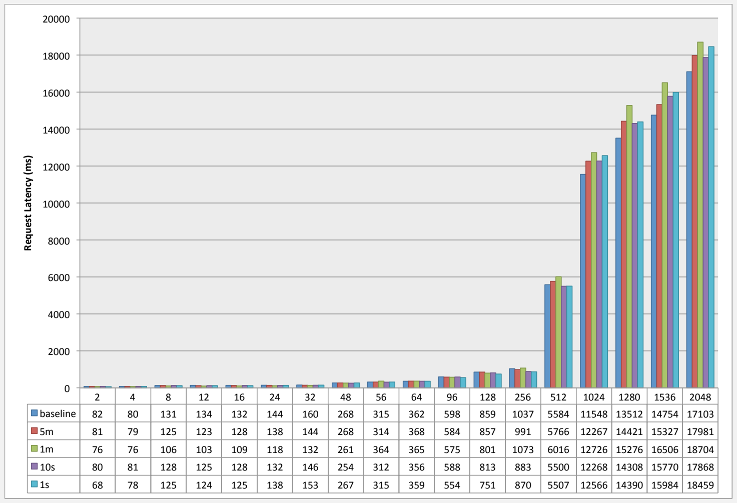


	2	4	8	12	16	24	32	48	56	64	96	128	256	512	1024	1280	1536	2048
🟦 baseline	94	224	450	727	1013	1586	2082	3241	3802	4068	6320	8017	9777	12697	16797	17989	19119	20388
🟥 5m	99	228	454	744	1043	1651	2314	3472	3943	4722	6506	9072	10370	13770	17895	19290	19812	21465
🟩 1m	103	243	594	989	1338	2096	2795	4166	4331	5431	8174	10364	11890	15466	19194	21436	21495	22838
🟪 10s	104	200	436	717	1042	1640	2221	3541	3963	4608	7095	8944	11116	13754	17954	18906	19894	21284
🟦 1s	98	206	447	786	1077	1736	2279	3523	4018	4760	6863	9600	11370	14357	18527	19508	20535	22152

Notes:

1. In the CPU-bound Fibonacci test, the biggest relative increase in request latency occurs between the runs with two and four threads. This correlates to the two physical CPU cores on the system exceeding capacity.
2. The 1m config run exhibits the worst performance of all the CPU-bound tests. This shows that global metric monitoring is actually more CPU intensive than collecting per-process metrics, even for processes with many active processes.
3. The rate at which request latency worsens begins to level off starting at the 512 thread load point. This is also the load point at which Apache begins to return errors. As the percentage of requests resulting in errors increases, the latency of the successful requests improves slightly. This explains the trend line shift.

Disk-Bound Results



	2	4	8	12	16	24	32	48	56	64	96	128	256	512	1024	1280	1536	2048
baseline	82	80	131	134	132	144	160	268	315	362	598	859	1037	5584	11548	13512	14754	17103
5m	81	79	125	123	128	138	144	268	314	368	584	857	991	5766	12267	14421	15327	17981
1m	76	76	106	103	109	118	132	261	364	365	575	801	1073	6016	12726	15276	16506	18704
10s	80	81	128	125	128	132	146	254	312	356	588	813	883	5500	12268	14308	15770	17868
1s	68	78	125	124	125	138	153	267	315	359	554	751	870	5507	12566	14390	15984	18459

Notes:

1. Apache's disk-bound performance measured in the 5MB download test is relatively unaffected by *dCAMP*. This is expected since the infrequent samples being logged to an output file are *dCAMP*'s only disk access.
2. The graph also shows the 512 thread load point as the beginning of a trend line shift, again correlating with the increase in request error rate.

Observations

1. When nodes are not expected to fail frequently, using longer heartbeat periods reduces the impact *dCAMP* has on the system.
2. It is better to monitor a process using a faster sample period than an entire system using a slower sample period.
3. The *dCAMP* system impact is noticeable but a considerably smaller factor than the impact hardware limitations have on performance monitoring.
4. Holding all else constant, slower sample periods have an obviously lower impact on system performance compared to faster sample periods.
5. Possibly using *dCAMP*'s reporting threshold, system impact can be minimized while still maintaining fine sample granularity.

Notes:

Strategy

One of the primary measures of scalability for a distributed system is its network traffic.[3] By simulating successively larger *dCAMP* systems (with respect to node count), one can extrapolate *dCAMP*'s effectiveness at monitoring large distributed systems and how to best configure its metric collections.

Notes:

Workload

dCAMP is setup to monitor a machine's global metrics, scaling the number of **simulated nodes in the dCAMP system from 3 nodes up to 200 nodes**. The metric configuration is kept constant for each test run. As *dCAMP* starts, monitors in steady state, and shuts down, the machine's network traffic is monitored and recorded every five seconds.

The test machine is a MacBook Pro (2.7GHz Core i7, 8GB 1333MHz DDR3, SSD) running OSX 10.9. All simulated *dCAMP* nodes use endpoints on the machine's loopback interface, and only the loopback interface traffic is monitored. The machine is otherwise entirely idle during the test runs.

Notes:

1. three-node system: one *Root*, one *Collector*, one *Metric*
2. 200-node system: eight groups with twenty-five nodes per group

Configuration

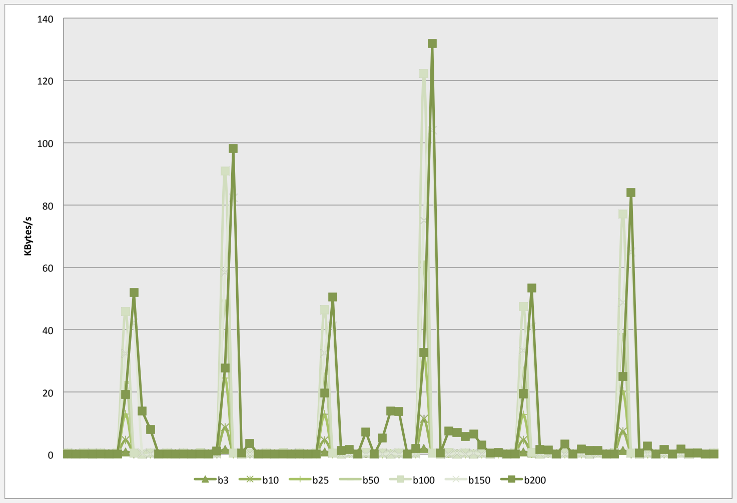
dCAMP is configured to monitor and report the below **global metrics**, using a heartbeat of 60 seconds.

- CPU usage every 60 seconds
- total disk throughput every 120 seconds
- total network throughput every 120 seconds
- memory usage every 60 seconds

Notes:

1. No thresholds were defined for the above configuration. That is, Sensor nodes immediately reported every sample instead of holding them for later reporting.

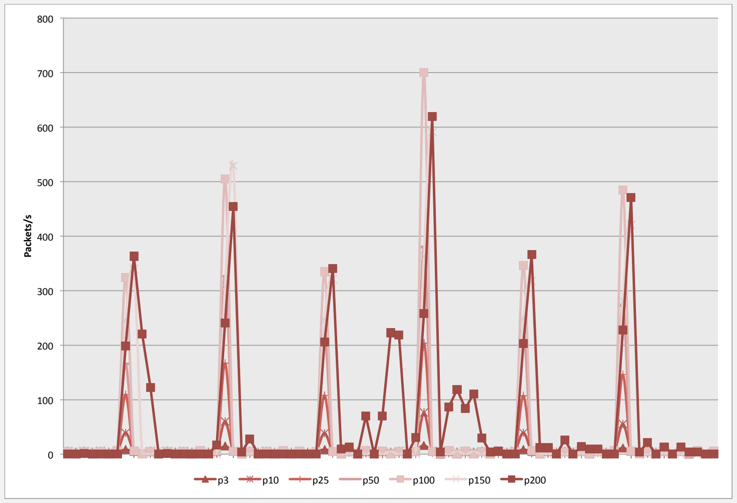
Steady State Network Bytes



Notes:

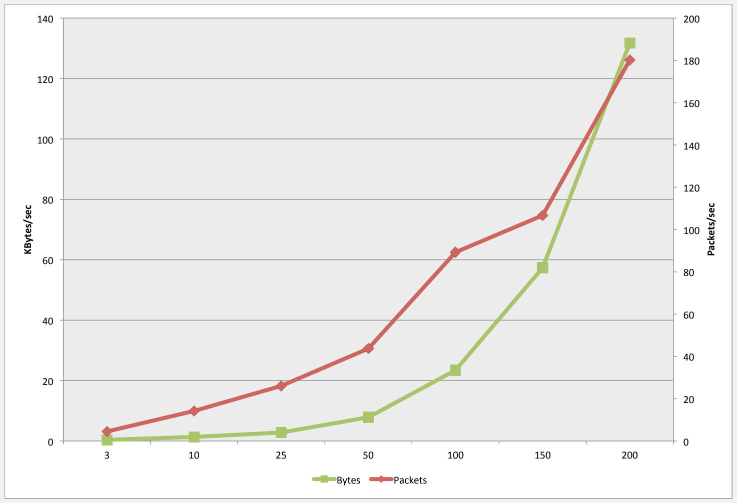
1. The rest (other than start-up/shut-down) of steady operation shows expected low network traffic except on sample periods.

Steady State Network Packets



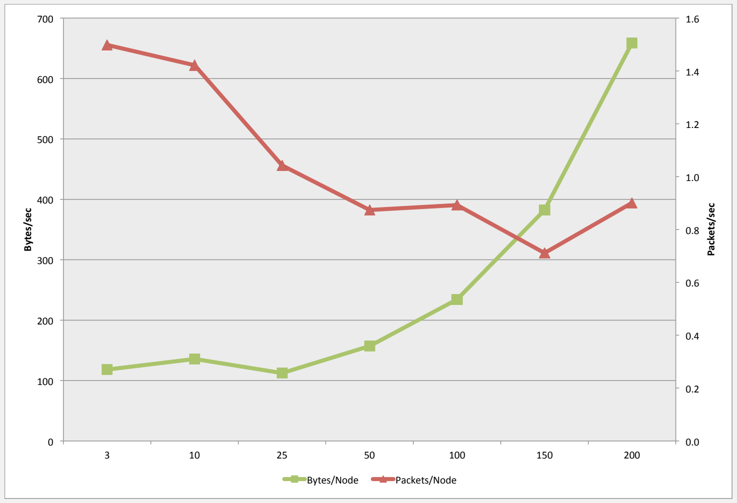
Notes:

Average Network Utilization



Notes:

Average Network Utilization Per Node



Notes:

Observations

1. Sparklines of each load point show the same pattern: highest network traffic occurs during start up and then also on shutdown.
2. As the node count increases, the rate at which bytes/packets are sent and received increases.
3. Comparing the same rates to the number of nodes in the system, one sees the configuration size grows faster than the number of nodes.
4. The number of messages being sent per node actually goes down and levels off just under 1 packet per node per second.
5. As the ratio of Sensor to Collector increases, the number of messages per node is expected to decrease. Therefore, a higher number of child nodes per parent should result in lower network utilization and better *dCAMP* scalability.

Notes:

1. traffic at start/end: follows from *dCAMP*'s use of chatty configuration protocols and a terse data protocol.
2. node count + net increase: correlates with the larger configuration which *dCAMP* must track as well as the additional nodes sending and receiving data.
3. bytes per node: more configuration data to pass around
4. packets per node: can be attributed to the fact that the number of Sensor nodes increases faster in relation to the number of *Collector* nodes.
5. That is, Sensor nodes do not require full-configuration replication and send/receive fewer messages since they are relatively uninvolved with topology coordination in comparison to *Collector* nodes.

Evaluation

The previous section presents an empirical evaluation of *dCAMP*'s **transparency** and **scalability**, showing *dCAMP* to be both transparent and scalable when properly configured.

Validity and **portability** are inherited from *CAMP* as well as the use of portable Python libraries. *dCAMP*'s **data delivery models** and **completeness** are apparent in the system's design and configuration capabilities, and these will be easily achieved with a full implementation of *dCAMP*.

Likewise, updating *dCAMP* to meet the **security** criterion is unfinished work, but the *dCAMP* design and use of ZeroMQ makes this work straightforward.

Notes:

1. The updated distributed performance framework criterion is introduced and used to evaluate *dCAMP*.
2. several improvements can be made to further *dCAMP*'s transparency and scalability

Contributions

In this work I presented the **Distributed Common API for Measuring Performance**, a distributed performance framework built on top of *CAMP* [1]. I described the design and implementation of *dCAMP*, using roles and services on top of ZeroMQ to build a simple, reliable distributed system.

I also presented a set of **criterion for evaluating distributed performance frameworks** by extending and updating the criterion presented by Zanikolas [3]. I used this criterion to evaluate *dCAMP* along with several other level 2 distributed performance frameworks and related works.

Notes:

1. again, level 2: dpf with at least one type of republisher in addition to producers

References

[Gabel, 2007] Gabel, Mark and Haungs, Michael (2007)
CAMP: a common API for measuring performance
LISA'07: Proceedings of the 21st conference on Large Installation System Administration Conference, p1–14

[Hintjens, 2011] Pieter Hintjens (2011)
Clustered Hashmap Protocol
`http://rfc.zeromq.org/spec:12/CHP`, April 2011

[Zanikolas, 2005] Zanikolas,, Serafeim and Sakellariou,, Rizos (2005)
A taxonomy of grid monitoring systems
Future Gener. Comput. Syst. 21(1), 0167-739X, p163–188.

Terminology

dCAMP Service: Services are a way of logically grouping functions within the *dCAMP* system. Each service is implemented in *dCAMP* as an independent thread.

dCAMP Role: Roles in the *dCAMP* system are groupings of one or more *dCAMP* services. There does not exist a one-to-one relationship between roles and services.

ZeroMQ: ZeroMQ is a message queuing framework which allows a developer to build distributed systems by focusing on the data and implementing simple design patterns.

ZeroMQ Address: A ØMQ address is the combination of network host identifier (i.e. an IP Address or resolvable name) and Internet socket port number.

ZeroMQ Endpoint: An endpoint is the combination of any ZeroMQ transport (`pgm`, `inproc`, `ipc`, or `tcp`) and a ØMQ address.

Terminology

Performance Metric: Performance metrics are any data about a given node relating to its throughput, capacity, utilization, or latency.

Metric Sampling: Metric collection or sampling is the process of measuring metrics on a given node.

Metric Reporting: Metric reporting is the process of sending sampled metrics to a parent node.

Metric Aggregation: Metric aggregation is the process of combining metrics from multiple nodes into a single metric, providing a coarser granularity for the performance metrics. Metrics are combined by calculating a sum, average, percent, or any other mathematically relevant operation.

Metric Calculation: Metric calculation is the process of combining identical metrics from multiple timestamps into a single metric.

Data Delivery Models

Monitoring information includes fairly static (e.g., software and hardware configuration of a given node) and dynamic events (e.g., current processor load, memory), which suggests the use of different measurement policies (e.g., periodic or on demand). In addition, consumer patterns may vary from sparse interactions to long lived subscriptions for receiving a constant stream of events. In this regard, the monitoring system must support both pull and push data delivery models.

Security

Certain scenarios may require a monitoring service to support security services such as access control, single or mutual authentication of parties, and secure transport of monitoring information.

Scalability

Monitoring systems have to cope efficiently with a growing number of resources, events and users. This scalability can be achieved as a result of good performance which guarantees that a monitoring system will achieve the needed throughput within an acceptable response time in a variety of load scenarios.

Transparency

Transparency refers to the lack of impact a distributed performance framework makes on the system being monitored. As [3] states, it is "typically measured as a function of host (processor, memory, I/O) and network load (bandwidth) generated by the collection, processing and distribution of events." If a framework lacks transparency it will fail to allow the underlying distributed system to perform well and will produce inaccurate performance measurements, thereby reducing its Scalability and destroying its Validity.

Completeness

The Completeness of a distributed performance framework refers to the exhaustiveness to which it gathers performance metrics. At a minimum, a framework must provide interfaces for measuring and aggregating performance data about a system's processor, memory, disk, and network usage. Several distributed performance frameworks provide further detailed performance metrics about the given distributed system being monitored, but this is usually at the cost of Portability.

Validity

A distributed performance framework is only as good as the data is produces; if the sensors or gathering techniques are inaccurate, then the data is useless at best, misleading at worst. Validity of a framework is achieved when the authors of a framework provide formal verification of its accuracy.

Portability

A framework's ability to run on a completely heterogeneous distributed system without special considerations by the practitioner is what this work defines as Portability. More specifically, a portable framework has a unified API regardless of the system architecture, does not restrict itself to applications written in specific programming languages, and does not require practitioners to manually instrument their application code. This black box characteristic is vital for a viable distributed performance framework's effectiveness as it allows practitioners to focus on the performance data and not on a myriad of APIs for various architectures or languages.

NetLogger

Their research has shown NetLogger to be highly scalable, complete, and transparent as well as valid. The activation service provides a push data delivery and can utilize the security mechanisms part of current web services in order to authenticate requests for performance data. NetLogger is currently implemented for C, C++, Java, Perl, and Python applications. Because the framework lacks black box characteristics, its **portability is greatly reduced**.

JAMM

JAMM, being heavily based off of NetLogger, inherits the validity, completeness, security, and transparency of NetLogger along with its lack of **portability**. JAMM does, however, prove itself in terms of scalability with it's own architecture.

Hawkeye

While no experiments have been run, the generally centralized manager reduces the Hawkeye framework's scalability, and its transparency is unknown. The frameworks module based producer architecture gives it an infinite completeness, but being only available on Linux and Solaris makes the framework less **portable**. Lastly, the ability to run jobs **securely** on target machines has been left as future work by the authors.

SCALEA-G

The framework makes use of secure sockets to achieve secure communications and achieves high completeness via code instrumentation. Unfortunately, the authors do not provide any report on SCALEA-G's **validity** or **transparency**.

SCALEA-G

The framework makes use of secure sockets to achieve secure communications and achieves high completeness via code instrumentation. Unfortunately, the authors do not provide any report on SCALEA-G's **validity** or **transparency**.

Host sFlow

In supporting host and application performance metric analysis alongside network metrics in one common system, sFlow has an advantage over more traditional host-only distributed performance frameworks. While sFlow's claims to scalable and accurate network level monitoring have been validated, less work has been to show the same for Host sFlow.

Ganglia

The analysis presented shows the design scales and maintains transparency for systems of several hundred nodes. Still, **scalability** is a concern of the authors since the multicast protocol exhibits a quadratic trendline as the number of nodes within a cluster increases. Memory usage and inter-cluster bandwidth also increase as the number of nodes increases, albeit much more linearly. In comparison, *dCAMP* memory usage is nearly constant since performance data is not persisted in memory to the same extent.

ZeroMQ in 100 Words

ØMQ (also known as ZeroMQ, 0MQ, or zmq) looks like an embeddable networking library but acts like a concurrency framework. It gives you sockets that carry atomic messages across various transports like in-process, inter-process, TCP, and multicast. You can connect sockets N-to-N with patterns like fan-out, pub-sub, task distribution, and request-reply. It's fast enough to be the fabric for clustered products. Its asynchronous I/O model gives you scalable multicore applications, built as asynchronous message-processing tasks. It has a score of language APIs and runs on most operating systems. ØMQ is from iMatix [`http://www.imatix.com`] and is LGPLv3 open source.

ZeroMQ Sockets

ØMQ sockets mimic standard TCP sockets, exposing interfaces for creating and destroying instances, binding and connecting to network endpoints, and sending and receiving data. However, they have two key differences from their TCP counterparts.

- asynchronous—the actual sending and receiving of data on a ZeroMQ socket is handled by a background thread.
- built-in support for one-to-many connections; a single socket can send and receive data from multiple endpoints.

ZeroMQ Messages

ZeroMQ messages are the building blocks of all data sent across ZeroMQ sockets. A message is comprised of one or more frames (or parts), and a single frame can be any size (including zero) that fits in memory. ZeroMQ guarantees messages are delivered atomically, meaning either all frames of the message are sent/received or none of the frames. Lastly, because sockets are asynchronous and messages are atomic, the entire message must fit in memory.

ZeroMQ Messaging Patterns

Publish-Subscribe: "The publish-subscribe pattern is used for one-to-many distribution of data from a single publisher to multiple subscribers in a fan out fashion."

Request-Reply: "The request-reply pattern is used for sending requests from a [...] client to one or more [...] services, and receiving subsequent replies to each request sent."

Pipeline: "The pipeline pattern is used for distributing data to nodes arranged in a pipeline. Data always flows down the pipeline, and each stage of the pipeline is connected to at least one node. When a pipeline stage is connected to multiple nodes data is round-robined among all connected nodes."

Exclusive Pair: "The exclusive pair pattern is used to connect a peer to precisely one other peer. This pattern is used for inter-thread communication across the inproc transport."

Metric Groups

In *dCAMP*, metrics are grouped into four different sets of performance metrics—global, network, disk, and per-process—and a fifth set of inquiry metrics.

Metric Details

Type	**Single Sample**	**Calculation**
basic	raw value at specified timestamp	$C = V_{t_2}$
delta	raw value at specified timestamp	$C = V_{t_2} - V_{t_1}$
rate	raw value at timestamp	$C = \frac{V_{t_2} - V_{t_1}}{t_2 - t_1}$
average	raw value and base value at timestamp	$C = \frac{V_{t_2} - V_{t_1}}{B_{t_2} - B_{t_1}}$
percent	raw value and base value at timestamp	$C = 100\frac{V_{t_2} - V_{t_1}}{B_{t_2} - B_{t_1}}$