DCAMP : DISTRIBUTED COMMON API FOR MEASURING PERFORMANCE

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Alexander Paul Sideropoulos

June 2014

COMMITTEE MEMBERSHIP

TITLE:	*dCAMP* : Distributed Common API for Measuring Performance
AUTHOR:	Alexander Paul Sideropoulos
DATE SUBMITTED:	June 2014
COMMITTEE CHAIR:	Dr. Michael Haungs
COMMITTEE MEMBER:	Dr. Aaron Keen
COMMITTEE MEMBER:	Dr. David Janzen

Abstract

dCAMP : Distributed Common API for Measuring Performance

Alexander Paul Sideropoulos

Although the nearing end of Moore's Law has been predicted numerous times in the past [15], it will eventually come to pass. In forethought of this, many modern computing systems to become increasingly complex, distributed, and parallel. As software is developed on and for these complex systems, a common API is necessary for gathering vital performance related metrics while remaining transparent to the user, both in terms of system impact and ease of use. Several distributed performance monitoring and testing systems have been proposed and implemented by both research and commercial institutions. However, most of these systems do not meet several fundamental criterion for a truly useful distributed performance monitoring system: 1) variable data delivery models, 2) security, 3) scalability, 4) transparency, 5) completeness, 6) validity, and 7) portability [20]. This work presents the Distributed Common API for Measuring Performance, *dCAMP* , as a solution to meet each of these criterion.

Acknowledgements

Thank you...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As the Internet has become more pervasive in today's business economy, there has been a natural trend of distributing large, complex systems across multiple components locally and throughout the world. These systems are not always homogeneous with respect to hardware architecture or even operating system, and development of these system can prove to be quite difficult even with the best tools available. In order to effectively build these systems, software engineers must be able to test their system for performance defects as well as bottlenecks. Additionally, distributed systems must respond to changes in availability and work load on its individual nodes.

Distributed performance testing frameworks supply software practitioners and system administrators with tools to evaluate the performance of a system from both black box and white box perspectives by publishing interfaces for instrumenting, collecting, analyzing, and visualizing performance data across the distributed system and distributed applications. Distributed performance monitoring frameworks, often considered part of the testing framework, provide a black box interface into monitoring a distributed system or application and usually includes mechanisms for triggering actions based on performance events. For the purpose of this work, I introduce the term distributed performance framework to collectively refer to both distributed performance testing and distributed performance monitoring frameworks.

1.1 Distributed Performance Framework Criterion

In order for practitioners and researchers alike to effectively use 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 [20] for grid systems. Data Delivery Models and Security have been taken directly from their work. Scalability has been modified to only consider good performance as its goal while Low Intrusiveness has been turned into Transparency. Extensibility has been removed from the list, and Completeness and Validity have been added. This work provides an alternate definition for Portability.

1.1.1 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. [20]

1.1.2 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. [20]

1.1.3 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. [20]

1.1.4 Transparency

Transparency refers to the lack of impact a distributed performance framework makes on the system being monitored. As [20] 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.

1.1.5 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.

1.1.6 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 inaccurate and useless. Validity of a framework is achieved when the authors of a framework provide formal verification of its accuracy.

1.1.7 Portability

A framework's ability to run on a completely heterogeneous distributed system without special considerations by the practitioner is what I define 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.

Chapter 2

dCAMP

The 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* [5]. The fundamental functionality of *CAMP* is providing an accurate and "consistent method for retrieving system performance data from multiple platforms." *dCAMP* takes advantage of this functionality and the authors' work done in validating *CAMP* 's accuracy and adds the following core feature sets:

- Stateful Performance API
- Distributed Performance Data Aggregation
- Performance Filters and Triggers
- Simplistic Fault Tolerance

2.1 Terminology

Knowing the following terminology will make it easier to understand and discuss the *dCAMP* project, its main goals, its usage, its components, and how it works.

Distributed Performance Testing Framework (DPTF),

Distributed Performance Monitoring Framework (DPMF),

Distributed Performance Framework (DPF): An DPTF or DPMF (collectively termed DPF) is a framework which allows its users to evaluate the performance of a system from both black box and white box perspectives by publishing interfaces for instrumenting, collecting, analyzing, and visualizing performance data across the distributed system and distributed applications. Typically, the framework provides a black box interface into monitoring a distributed system or application and includes mechanisms for triggering actions based on performance events. The *dCAMP* project is designed to be a DPF.

Performance Metric,

Performance Counter: Performance metrics are any data about a given node relating to its throughput, capacity, utilization, or latency. In *dCAMP* , these are grouped into four different sets of performance metrics (global, network, disk, and per-process) and a fifth set of inquiry metrics. They are described fully in section 2.2.

Metric Aggregation: Metric aggregation is the process of combining metrics from multiple nodes into a single metric. Performance metrics, while useful at an individual system granularity, can be rather limited in value for a DPF where the goal is measurement of the distributed system as a whole. Metric aggregation provides a coarser granularity for the performance metrics, calculating a sum, average, percent, or any other mathematically relevant operation across multiple nodes in the system.

Metric Calculation: Metric calculation is the process of combining identical metrics from multiple timestamps into a single metric. Various equations and inputs used to do this calculation, chosen depending on the type of metric and desired representation; these equations are listed in Table 4.1.

Filter,

Throttle,

Threshold: Filtering (or throttling or thresholding) provides a mechanism for reducing the amount of data sent between nodes of the system. Filtering allows a user to specify when or at what point to report metrics from one level to its parent. For example, a filter might be set to only report average CPU utilization that is over seventy-five percent.

***dCAMP* Node,**

***dCAMP* Process:** A single, independently running instance of *dCAMP* in the distributed system is called a *dCAMP* node or process. More than one node may exist on a single computer. A node consists of the Node role and zero or more other *dCAMP* roles.

***dCAMP* Service:** Services are a way of logically grouping functions within the *dCAMP* system, from performance metric sampling to *dCAMP* system management. A description of all the *dCAMP* services can be found in section 3.1. 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; the *dCAMP* role-to-service mapping can be seen in Table 3.1.

***dCAMP* Hierarchy:** The *dCAMP* system is organized in a hierarchical pattern with respect to data movement and system control functionality. The hierarchy can be thought of as a tree structure, with leaf nodes being at the top of the hierarchy and a single root node at the bottom. Metric data moves down the hierarchy from leaves to the root; configuration data and control commands move up from the root to the leaves.

***dCAMP* Level:** Levels are a way of organizing the *dCAMP* hierarchy horizontally. Levels are defined by their distance from the root node. For example, level one is one node away from the root node, or said another way, the first level is directly "connected" to the root node. The second level is two nodes away from the root node, or any node in the second level is connected to the root node by another node (in the first level). This necessarily means the root is in level zero (all by itself).

Parent Node: Nodes are called parent nodes if there exists at least one node connected to it from a level of higher ordinal value. For example, a node in level one with at least one node connected to it from level two is considered a parent node. The root node is inherently a parent node.

Child Node: A node is called a child node if it is connected to another node in a level of lower ordinal value. For example, a node in level one is connected to the root node (in level zero), so it is called a child node. The root node is the only node in the *dCAMP* system which is not a child node.

***dCAMP* Configuration:** The *dCAMP* configuration specifies everything about the system, including hierarchy levels, metrics, sampling periods, reporting periods, filtering, communication details, etc. The configuration is set at the root node and then distributed to the rest of the *dCAMP* system. Configuration details can be found in section 2.3.

ZeroMQ Address,

ZeroMQ Endpoint: A ØMQ address is the combination of network host identifier (i.e. an IP Address or resolvable name) and Internet socket port number. An endpoint is then the combination of any ZeroMQ transport (`pgm`, `inproc`, `ipc`, or `tcp`) and an address.

Metric Collection,

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.

2.2 *dCAMP* Metrics

Metrics marked with "(*dCAMP*)" are extensions added by the *dCAMP* project to the basic CAMP metrics. These provide a performance view of multiple nodes in the distributed network and are collected by the Aggregate service rather than the CAMP service.

2.2.1 Global Metrics

Global metrics measure overall CPU, process, thread, and memory usage of the system.

- Node CPU usage
- Node free physical memory (KB)

- Aggregate average CPU usage (*dCAMP*)
- Aggregate free physical memory (percentage) (*dCAMP*)

2.2.2 Network I/O Metrics

Network metrics measure utilization of a given network interface on the system.

- Total bytes sent on the given interface
- Total packets sent on the given interface
- Total bytes received on the given interface
- Total packets received on the given interface
- Aggregate bytes sent (*dCAMP*)
- Aggregate packets sent (*dCAMP*)
- Aggregate bytes received (*dCAMP*)
- Aggregate packets received (*dCAMP*)

2.2.3 Disk I/O Metrics

Disk I/O metrics measure throughput of a given disk or partition on the system.

- Number of read operations on the given disk
- Number of write operations on the given disk
- Number of read operations on the given partition
- Number of write operations on the given partition
- Aggregate number of read operations (*dCAMP*)
- Aggregate number of write operations (*dCAMP*)

2.2.4 Per-process Metrics

Per-process metrics measure CPU, memory, and thread usage for a single process on the system.

- Number of major and minor page faults
- Process CPU utilization
- Process user mode CPU utilization
- Process privileged mode CPU utilization
- Size of the process' working set in KB
- Size of the used virtual address space in KB
- Number of threads contained in the process

2.2.5 Inquiry Metrics

Inquiry metrics provide a mechanism for enumerating various properties of the system.

- Enumeration of the available disk partitions
- Enumeration of the available physical disks
- Enumeration of the valid inputs to the network functions
- Number of CPUs in the system
- "process identifier" for the given pid
- "process identifier" for each running process launched from an executable of the given name

2.3 Configuration

2.3.1 Node Specification

Nodes may be specified individually (as ZeroMQ addresses) or as groups (IP subnets). Additionally, nodes may be included or excluded based on host name or IP Address matching. Name matching does a case-insensitive comparison of the node's host name; left, right, or whole name matching can be specified. Address matching checks that the node's IP Address falls within a given subnet (i.e. IP Address and mask length).

```
node-spec       = address / node-group
address         = host ":" port
host            = name / ip-address
node-group      = group-name 1*( address / ( subnet ":" port ) ) *f
subnet          = ip-address "/" mask-length
filter          = [ "+" / "-" ] ( name-match / subnet-match )
name-match      = [ ( "L" / "R" / "W" ) SP ] name
subnet-match    = subnet
```

Figure 2.1: Configuration File - Node Specification

2.3.2 Sample Specification

Performance metric samples are specified as:

1. the node(s) on which to sample the data,
2. the rate at which data should be sampled,

3. the threshold past which data should be reported, and lastly
4. the actual performance metric to be sampled.

The report threshold can be specified as "hold and report every N seconds" or "report when the metric value is greater/less than X". When "hold" is specified (via an $*$), all metric values sampled during the time limit are sent.

```
sample-spec        = 1*sample
sample             = sample-rate [ report-threshold ] metric
sample-rate        = 1*DIGIT "s" ; seconds
report-threshold = ( "*" 1*DIGIT "s" ) / ( ( "<" / ">" ) 1*DIGIT ) ;
metric             = "CPU" / "DISK" / "NETWORK"
```

Figure 2.2: Configuration File - Sample Specification

NOTE ABOUT ACCUMULATION (DOESN'T WORK)

- filtering could be done two ways: accumulatively and discretely.
- accumulative means we send only one final value for each time range (e.g. collect every second but report every minute, so 60 samples are combined into a single value and sent)
- discrete means we send each constituent value for each time range, but they are "held" until the time limit is reached
- how do these two filtering methods interact with value-based limits? are they always discrete?
- ACTUALLY: accumulation is not valuable for monotonically increasing values–it is the same as just sampling at the slower frequency. accumulation is only valuable for non-monotonically increasing values. but in that case, one should find the raw, monotonically increasing values from which

it is calculated.

Chapter 3

Design

General design: semi-centralized, hierarchical peer-to-peer system utilizing the pipe-flow architecture pattern in which leaf (sensor) nodes of the hierarchy collect data, filter out extraneous data, and send it up the pipe to an aggregate node which subsequently filters out more data and sends it up to another aggregate or root node.

dCAMP was designed to be simple and only add complexity where needed. This allows, for example, for quick and easy, large scale testing. Additionally, in order to be transparent, minimizing network traffic was important.

3.1 *dCAMP* Roles and Services

The *dCAMP* distributed system is comprised of one or more nodes each running one or more roles. Each role–a published, remotely accessible interface–provides one or more sets of functionality. Each set of functionality is known as a service.

3.1.1 Services

- **Node**—rudimentary *dCAMP* functionality; handles topology communication, heartbeat monitoring, and failure recovery.
- **Sensor**—local performance metric gathering; essentially the *dCAMP* layer on top of the OS and hardware performance APIs (accessed via CAMP).
- **Filter**—performance metric filtering; provides throttling and thresholding of metrics.
- **Aggregation**–performance metric aggregation; provides collection of and calculation on metrics from multiple sensors and/or collectors.
- **Management**–primary entry-point for end-user control of *dCAMP* distributed system; this is the *dCAMP* instrument panel, providing basic administration functions (e.g. start, stop, etc.).
- **Configuration**–complete or partial configuration replication; provides topology and configuration distribution.

3.1.2 Roles

The *Base* role must be running on each node for it to be part of the *dCAMP* distributed system. 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.

The *Metric* role runs on the nodes from which performance metrics should be collected. The *Collector* role acts as an aggregation point in the system, combining performance data from multiple *Metric* (and *Collector*) nodes and providing additional aggregated performance metrics.

There is only one *Root* role active in the system; it acts as the master copy

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

Table 3.1: Role to Service Mappings

of the *dCAMP* configuration and sole user-interface point. The *Root* role is not strictly attached to any given node in the system. Rather, the *Root* role may dynamically move to any first-level *Collector* node if the current *Root* node fails.

Depending on the use case and desired system performance, an administrator may choose to split roles across multiple nodes or collapse them onto a single node. For example, a single node may act as *Metric*, *Collector*, and *Root* for smaller systems while larger systems would employ dedicated *Collector* nodes.

3.1.3 Role-to-Service Mappings

Table 3.1 lists the roles which can be "published" by a *dCAMP* node and the services which they implement.

3.2 Fault Tolerance

3.2.1 Heartbeating (Detecting Disconnections)

Disconnections are detected via a lack of messages, e.g. missed X consecutive messages or no messages received after D seconds.

- Metric nodes MUST detect when their parent (Collector) node disconnects. (Promotion Algorithm)

- The Root node MAY detect when a Collector node disconnects. (Promotion Algorithm)
- Collector nodes MUST detect when the Root node disconnects. (Election Algorithm)
- The Root node MUST detect when a Metric node rejoins the system. (Reminder Algorithm)

LATER

- should the configuration TTL be used somehow?
  - alternatively, this could be based on configuration entry TTL expiration.
  - collector-to-collector heartbeating? not needed, since these TTLs will come from the root (via the data protocol)
  - ttl used to know which first-level collectors are root-eligible?
- heartbeats are sent from first-level collectors to root or all collectors?
- does a collector need to know when a metric node disconnects? no.

3.2.2 Reminder Algorithm (Metric Node Recovery)

Metric nodes can leave and enter the *dCAMP* system at any time. When they rejoin, they should be placed back into the same location within the topology so as to maintain as much consistency within the performance data as possible.

The crux of this algorithm is the group definitions within the *dCAMP* configuration: nodes are always defined to be within a group, and the groups define the network topology. Essentially, this algorithm is incorporated into the Topology protocol; no additional work is necessary.

1. Sensor node rejoins the system with `POLO` response to Root node's `MARCO`

message.

2. Root node detects Metric node is already part of *dCAMP* system.
3. Root node (re)sends `ASSIGN` message to Metric node.

DETECTION

Detecting when a Metric node disconnects is not necessary. Rather the Root node only needs to detect when a Metric node rejoins the *dCAMP* system, comparing the Metric node's UUID to the UUID already saved in the topology.

QUESTIONS

- How does this account for new nodes which have been assigned to Collector node since this Metric node disconnected? Can the Collector node become overloaded, or is this already accounted for with the configuration groups?

3.2.3 Promotion Algorithm (Collector Node Recovery)

As with the Reminder Algorithm, this recovery relies heavily on the Topology Management Protocol. When a Metric node, M, detects its Collector node, C, is down,

1. M sends an `SOS` message to the Root node, R.
2. If R has received an `SOS` message from more than 1/3 of C's group, the algorithm proceeds as per below.

When the Root node, R, detects one of the Collector nodes has disconnected,

1. R broadcasts a `STOP` message to all nodes within C's group and clears the groups configuration from the *dCAMP* system.

2. R then broadcasts a `MARCO` message and begins rebuilding the group topology via the Topology protocol.

DETECTION

M will use the Configuration Replication Protocol (`HUGZ`) to detect when C disconnects. R will use the Data Flow Protocol (`DATA(type='HUGZ')`) to detect when any of its Collector nodes has disconnected. That is, if M or R receives no messages from C within D seconds, C is considered disconnected.

3.2.4 Election Algorithm (Root Node Recovery)

This recovery algorithm is based on the bully algorithm[1]. Only Collector nodes participate in the election, initiated when a Collector node, C, detects the Root node, R, is down.

1. C sends `WUTUP` message to all nodes whose UUID is higher than its own, expecting a `YO` message in response.
2. If C does not receive any `YO` messages,
   (a) C declares victory by sending `IWIN` message to all nodes, and
   (b) C waits W seconds before transitioning to become the Root, allowing for another node to replace it as Root via a separate election.
3. If C receives a `YO` message,
   (a) C waits for W seconds to receive an `IWIN` message from another node whose UUID is higher than its own.
   (b) If no `IWIN` message is received, C resends its `WUTUP` message and goes through the election process again.

Additionally,

- If C receives a `WUTUP` message from a node whose UUID is lower than its own, C responds with a `YO` message and then starts its own election.
- If C receives an `IWIN` message from a node whose UUID is lower than its own, C immediately begins a new election.

DETECTION

C will use the Configuration Replication Protocol (`HUGZ`) to detect when R disconnects. That is, if C receives no message from R within D seconds, R is considered disconnected.

QUESTIONS

- Is there a difference between starting a new election and resending the initial `WUTUP` message? Is there an optimization to be found by uniquely ID'ing each election?
- Can this algorithm be optimized w.r.t. the number of messages sent?

Chapter 4

Implementation

4.1 Requirements

4.1.1 Functional

1. Configuration
    - instantiation, administration
    - topology coordination
    - metric collection spec.
2. Metric Collection
    - *dCAMP* API on top of CAMP
    - filters (at various levels), thresholds
    - aggregation of metrics across nodes
    - output to log file
3. Fault Tolerance
    - simple rules to handle failures

4.1.2 Fault Tolerance

1. Topology MUST sustain brief network disconnectivity of any node.
2. Sensor nodes MUST be allowed to enter/exit topology at any time.
3. Root/Collector nodes (i.e. parents) MUST failover in case of extended disconnectivity.
   - Loss of previously collected data SHOULD be minimized during failover.
4. Management node SHOULD be allowed to enter/exit topology at any time.

4.1.3 Testing

- **Transparency:** *dCAMP* SHOULD introduce negligible performance impact on sensor nodes.
- **Accuracy:** *dCAMP* MUST accurately report metrics/performance of sensor nodes (individual and aggregated).
- **Scalability:** *dCAMP* SHOULD maintain its transparency and accuracy as it scales (i.e. the number of sensor nodes increases).
- **Fault Tolerance:** *dCAMP* MUST successfully handle entrance/exit of any node(s) in the system.

4.2 *dCAMP* Operation

4.2.1 Sequence of *dCAMP* Operation

The following steps describe how the *dCAMP* system is turned on. The Base nodes (other than the node assigned to be the Root) can be started at any time by using the *dCAMP* CLI, before or after the Root node is initialized. It is expected these Base nodes are managed by a watchdog utility which automatically restarts the node if it exits for any reason.

```
#!/usr/bin/env bash
while [ true ]
do
    dcamp base --address localhost:56789
done
```

Figure 4.1: Sample Watchdog Script

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.
    - Performance counters are sampled, filtered, reported, and logged by the Metric nodes at regular intervals according to the *dCAMP* Configuration.
    - Performance counters received from child nodes are aggregated, filtered, reported, and logged by Collector nodes at regular intervals according to the *dCAMP* Configuration.
    - 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).
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.

4.2.2 Threading Model

As mentioned above as the first and third steps of *dCAMP* operation, a Base node can transform into one of the three active *dCAMP* roles: Root, Collector, or Metric. This transformation is actually the Base role (via the Node service) launching and managing another Role internally. This interaction is depicted in Figure 4.2.

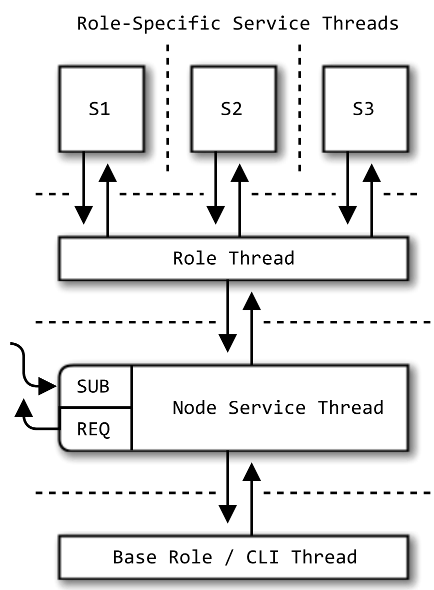


Figure 4.2: Node, Role, Services Threading Model Diagram: Thread boundaries are represented by dashed lines. Except for the Node service's `SUB` and `REQ` sockets, all arrows represent `PAIR` socket communication.

When a Base node is running, only the bottom two threads (the Base role and the Node service) are active. Once it receives an assignment from the "discover"

Topology Protocol or the *dCAMP* CLI, the Node service launches an appropriate Role thread which, in turn, launches one or more Role-specific service threads.

All communication between the roles and services occurs across `PAIR` control sockets. There are also various service-to-service communications which occur via `inproc` transport sockets (e.g. the internal Data Flow Protocol) and shared memory data structures (e.g. the Configuration service).

Also mentioned in section 4.2.1 as the last two steps, each Role exits and, by doing so, reverts itself back to a Base node. This is handled just like before, with the Node service receiving a `STOP` message via the "stop" Topology Protocol and then notifying the internally running Role to shut down. The Role thread then notifies its service threads, waits for them to finish, then exits.

4.3 ZeroMQ Protocols

For a quick background on ZeroMQ socket types and message patterns, please see Appendix A.

4.3.1 Topology Protocols

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.

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. 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.

```
discover = *R-MARCO B-POLO ( R-ASSIGN / R-WTF )
stop     = R-MARCO B-POLO ( R-STOP / R-WTF )
```

Figure 4.3: Topology Protocols: `R-` represents the Root node sending a message and `B-` represents a Base node sending a message.

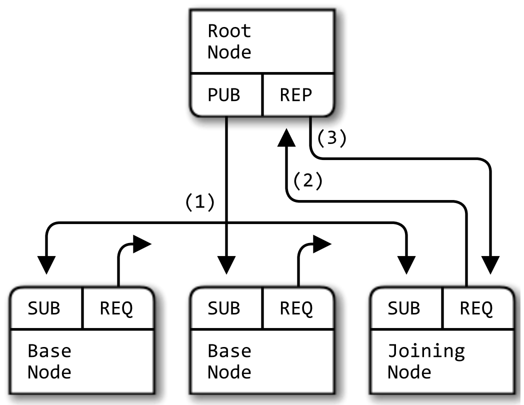


Figure 4.4: Topology Protocol Diagram: (1) Root sends `MARCO` at regular intervals, (2) Base sends `POLO` request, (3) Root replies with `ASSIGN` or `STOP`

Message Definitions

`TOPO` is a generic topology message consisting of four frames. This message type is designed to be sent across a PUB/SUB connection, from which subscribers filter incoming messages using the first frame. This design proves useful for the Recovery Protocols.

The `MARCO` message is simply shorthand for `TOPO(key="/MARCO")`.

```
Frame 0: key, as OMQ string
Frame 1: root address, as OMQ string
Frame 2: root UUID, 16 bytes in network order
Frame 3: <empty> or content, as OMQ string
```

Figure 4.5: `TOPO` **Message Definition**

`CONTROL` is a generic control message consisting of four frames and designed to be sent across a REQ/REP connection. The `POLO`, `ASSIGN`, and `STOP` messages are shorthand for `CONTROL(command="POLO")`, `CONTROL(command="ASSIGN")`, and `CONTROL(command="STOP")` respectively.

In the case of `ASSIGN`, the third frame contains the specific topology instructions (level-one collector, leaf node, etc.) being sent to the Base node.

```
Frame 0: command, as OMQ string
Frame 1: base address, as OMQ string
Frame 2: base UUID, 16 bytes in network order
Frame 3: properties, JSON-encoded, as OMQ string

command     = "polo" / "assignment"
properties  = *( parent / level / group )
parent      = "parent=" <node-address>
level       = "level=" ( "root" / "branch" / "leaf" )
group       = "group=" <group-identity>
```

Figure 4.6: `CONTROL` **Message Definition**

`WTF` is *dCAMP* 's error message type. It has three frames (though Frame 2 may be empty) with the first designed to make error detection simple.

```
Frame 0: "WTF", as OMQ string
Frame 1: error code, 4 bytes in network order
Frame 2: <empty> or error message, as OMQ string
```

Figure 4.7: `WTF` **Message Definition**

4.3.2 Configuration Replication Protocol

dCAMP configuration and topology state are replicated across the system using key-value pairs, with the keys laid out in a hierarchical fashion. This lends itself nicely to PUB/SUB topic filtering.

For example, because a Metric node only needs the configuration values for its particular group, the node subscribes only to the `"/CONFIG/<group-name>/"` topic. Any `KVPUB` whose key does not start with this string is then discarded.

In practice, Metric nodes need more than just their group-specific configuration, but the general principle holds true: nodes only receive the configuration data they require and nothing more. In the case of first-level Collector nodes, they receive all updates since they are fail-over candidates for the root node.

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

Figure 4.8: Configuration Protocol Specification: `P-` represents the parent node (Root or Collector) sending a message and `C-` represents the child node (Collector or Metric) sending a message.

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. 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. 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.

The *dCAMP* configuration replication algorithm adheres to the Clustered Hashmap Protocol[10] with a few minor (and one major) modifications:

1. only the Root node may write updates to the configuration,
2. the full configuration table will be replicated across all first-level Collector nodes (lower-level nodes may filter their configuration to only store relevant data),
3. a different set of command names are used (as described below), and
4. configuration updates are distributed via the *dCAMP* hierarchy (instead of directly from the Root node).

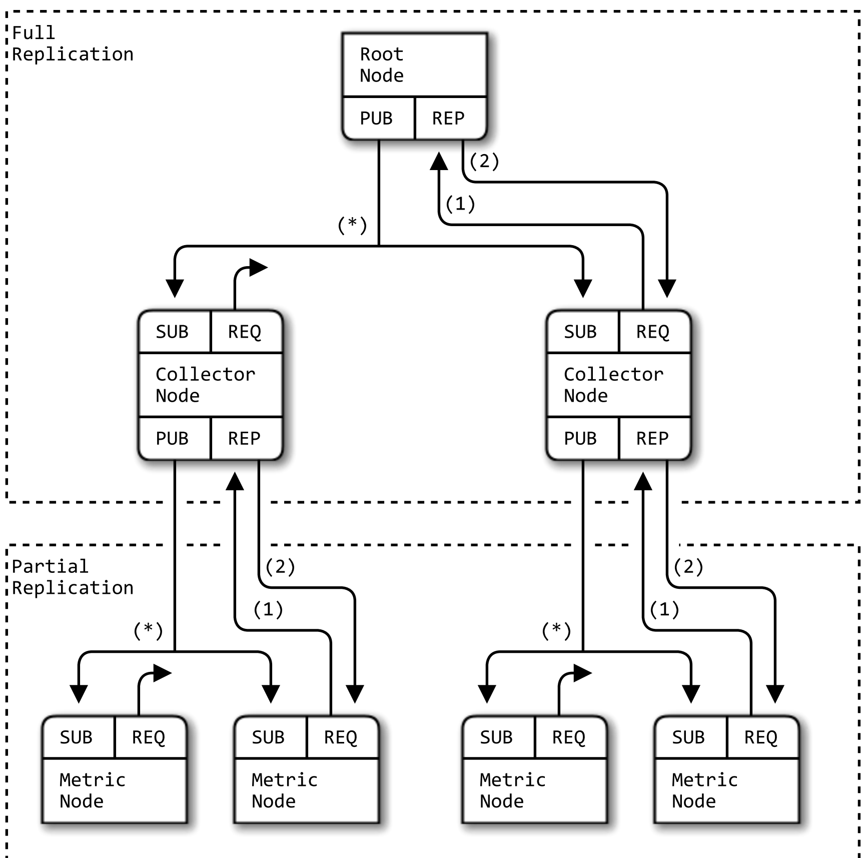


Figure 4.9: Configuration Protocol Diagram: (*) Parent node sends `KVPUB` or `HUGZ` at any time, (1) child node sends `ICANHAZ` request, (2) parent node replies with zero or more `KVSYNC` messages followed by exactly one `KTHXBAI` message.

Message Definitions

These messages come from the CHP protocol. Additionally, a `WTF` error message may be sent by the parent in case of error. It should be noted, each of the following messages is really the same five-frame format with varying keys and semantics.

As shown in Figure 4.8, the `ICANHAZ`, `KVSYNC`, and `KTHXBAI` messages are sent across a REQ/REP connection type. `KVPUB` (as the name would imply) along with the `HUGZ` heartbeat message are designed for the PUB/SUB pattern.

`ICANHAZ` is a configuration snapshot request sent by the child node when it first starts. Multiple `ICANHAZ` requests can be sent for the different topics or subtrees needed by the node, and the node will not begin normal operation until all of the requested values have been received.

```
Frame 0: "ICANHAZ", as 0MQ string
Frame 1: <empty>
Frame 2: <empty>
Frame 3: <empty>
Frame 4: subtree specification, as 0MQ string
```

Figure 4.10: `ICANHAZ` **Message Definition**

`KVSYNC` is a configuration snapshot response message. For every key-value pair within the requested subtree, a `KVSYNC` message is sent to the child node. Note: if no values exist for a requested subtree, a `KTHXBAI` message will be the only response received by the child node.

The sequence number in Frame 1 SHOULD be ignored by the recipient since no order guarantees exist for configuration snapshots requests.

```
Frame 0: key, as 0MQ string
Frame 1: sequence number, 8 bytes in network order
Frame 2: <empty>
Frame 3: <empty>
Frame 4: value, as blob
```

Figure 4.11: `KVSYNC` **Message Definition**

`KTHXBAI` marks the end of a successful snapshot request. Frame 4 MUST contain the highest sequence number of all the values in the configuration snapshot.

```
Frame 0: "KTHXBAI", as 0MQ string
Frame 1: sequence number, 8 bytes in network order
Frame 2: <empty>
Frame 3: <empty>
Frame 4: subtree specification, as 0MQ string
```

Figure 4.12: `KTHXBAI` **Message Definition**

`KVPUB` is a configuration update sent from parent to child. The sequence number in Frame 1 must be monotonically increasing. When a `KVPUB` is received which has a sequence number lower than a previously received `KVPUB`, the node MUST delete its saved configuration values and request a new snapshot.

Frame 2 SHOULD contain the UUID of the node from which the value originated. In *dCAMP* , this should only be the Root node's UUID. Frame 3 MAY contain additional properties for the key-value pair, such as an ephemeral time-to-live.

```
Frame 0: key, as 0MQ string
Frame 1: sequence number, 8 bytes in network order
Frame 2: UUID, 16 bytes in network order
Frame 3: properties, JSON-encoded, as 0MQ string
Frame 4: value, as blob
```

Figure 4.13: `KVPUB` **Message Definition**

`HUGZ` is the heartbeat message sent from parent to child when the rate of `KVPUB` messages being sent drops below a predetermined threshold. The `HUGZ` message is critical to maintaining topological consistency in *dCAMP* .

```
Frame 0: "HUGZ"
Frame 1: 00000000
Frame 2: <empty>
Frame 3: <empty>
Frame 4: <empty>
```

Figure 4.14: `HUGZ` Message Definition

4.3.3 Data Flow Protocol

There are two data flow protocols in the *dCAMP* system: the external protocol for data flowing from one node to the next (via PUB/SUB) and the internal protocol for data flowing between components of a single node (via PUSH/PULL). Both protocols have the same specification and use the same message formats.

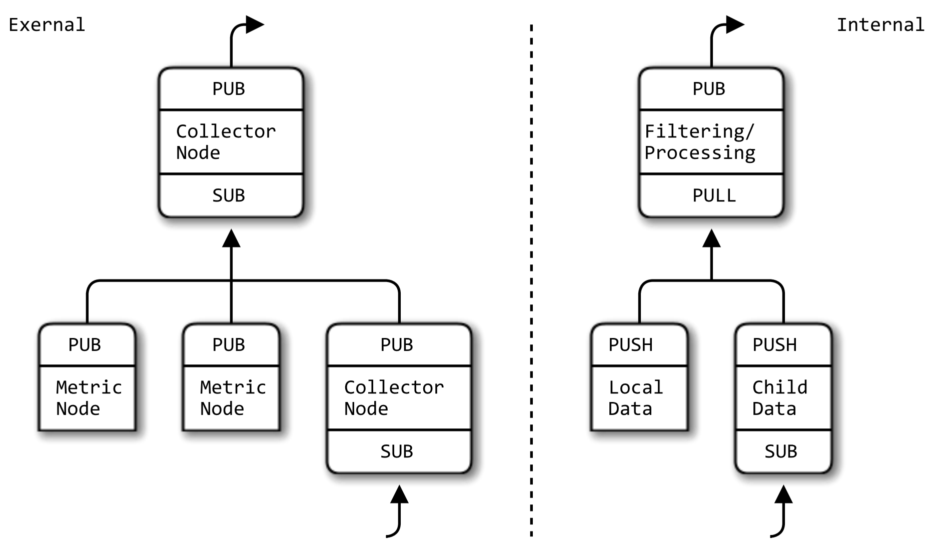


Figure 4.15: Data Flow Diagram

The *dCAMP* data flow protocol is very simple, comprised of a single data message type. The data flows from one node to another via PUB/SUB sockets. Internally, data flows from the upstream data producers, through a filtering/processing unit, and out to downstream data consumers via PUSH/PULL sockets.

When data rate is slower than a predefined threshold, heartbeats are sent instead to keep inter-node connections alive.

```
data-flow = *( DATA / HUGZ )
```

Figure 4.16: Data Flow Specification: All messages are sent from child (Metric or Collector) to parent (Collector or Root).

Performance Measurement

When discussing performance measurement, it is important to understand how metrics are sampled, calculated, and presented to an end user.

Performance metrics, also called counters, are usually monotonically increasing values. That is, reading its raw, instantaneous value is virtually meaningless; to correctly read the counter it must be sampled at two different points in time and then calculated.

For example, when displaying a graph of data point for non-basic metric types, each data point is really a calculated value of the value at the current timestamp and that at the previous timestamp. It is possible to look at fewer data samples to first get a course-grain view (e.g. five-minute samples) of the metric before drilling in a looking at finer-grain samples (e.g. one-second samples).

Non-monotonically increasing counters do exist (e.g. disk speed, Ethernet uplink speed, etc.), but these are usually fairly static configuration values and

Type	Contents of Single Sample	Calculation of Two Samples
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	$C = 100\frac{V_{t_2} - V_{t_1}}{B_{t_2} - B_{t_1}}$

Table 4.1: Metric Types: C represents the value calculated from two samples taken at t_1 and t_2. V is the value and B is the base value in the `DATA` message

do not need to be sampled frequently. *dCAMP* supports these types of counters with the “basic” metric type.

Table 4.1 shows how each of the *dCAMP* metric types are calculated. Note: unlike some other performance measurement frameworks[6], *dCAMP* stores all metrics in their raw, uncalculated form and only presents a calculated value upon display.

Message Definitions

`DATA` is a five-frame message containing the performance metric data sampled by the Sensor service or calculated by the Aggregation service. The `HUGZ` message is simply shorthand for `DATA(type="HUGZ")`.

A single data sample MUST contain: source identifier (node or aggregation), metric identifier, timestamp, and one or two values depending on the metric type.

In case of `HUGZ`, no other property strings are used, and Frames 3 through 5 are all empty. Frame 4 will be non-empty for average and percent types.

NOTE: This message needs to be cleaned up...its a bit too verbose. I think just the config-seqid is needed to identify the metric being sampled.

```
Frame 0: data source (leaf or collector node address), as 0MQ string
Frame 1: properties, JSON-encoded as 0MQ string
Frame 2: time in ms epoch utc, 8 bytes in network order
Frame 3: value, 8 bytes in network order
Frame 4: base value, 8 bytes in network order; only for average and percent types

properties = *( type / detail / config / seqid )
type       = "type=" ( "HUGZ" / "basic" / "delta" / "rate" / "average" / "percent"
detail     = "detail=" <string>
config     = "config-name=" <string>
seqid      = "config-seqid=" <integer>
```

Figure 4.17: `DATA` Message Definition

4.3.4 Recovery Protocols

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

```
branch-recovery  = *sos group-stop
sos              = M-SOS R-KEEPCALM
group-stop       = R-GROUP M-POLO R-STOP
```

Figure 4.18: Branch Recovery Protocol: `R-` represents the Root node sending a message and `M-` represents a Metric node sending a message.

The Branch Recovery Protocol is initiated by Metric nodes when they detect their Collector has died. 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. Once shut down, a new Collector is selected and the branch is rebuilt using the standard "discover" Topology Protocol.

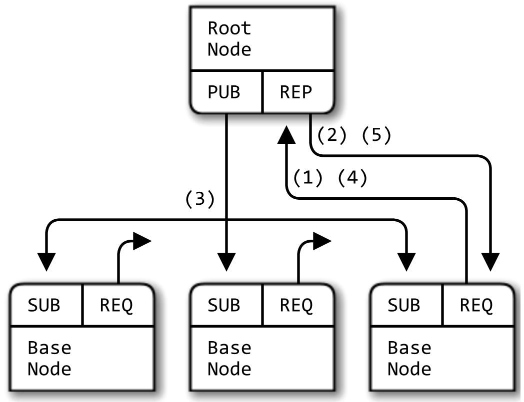


Figure 4.19: Branch Recovery Protocol Diagram: (1) Metric nodes send `SOS` requests, (2) Root replies with `KEEPCALM`, (3) Root sends `GROUP` only to nodes in branch, (4) Metric nodes send `POLO` requests, (5) Root replies with `STOP`

`SOS` and `KEEPCALM` are shorthand for the `CONTROL` message with a command value of `"sos"` and `"keepcalm"` respectively. The `POLO` and `STOP` messages come directly from the Topology Protocol.

The `GROUP` message is similarly shorthand for the `TOPO` message with a key value of `"/GROUP/<group-name>"`. This takes advantage of ZeroMQ's Pub-Sub filtering to only stop the faulty branch.

```
root-recovery  = *election
election       = C-WUTUP *C-YO C-IWIN
```

Figure 4.20: Root Recovery Protocol: `C-` represents a Collector node sending a message.

As each Collector node detects the Root node has died, it attempts to start an election via the `WUTUP` message. Collector nodes with higher UUIDs will respond to the first Collector by sending the `YO` message. 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.

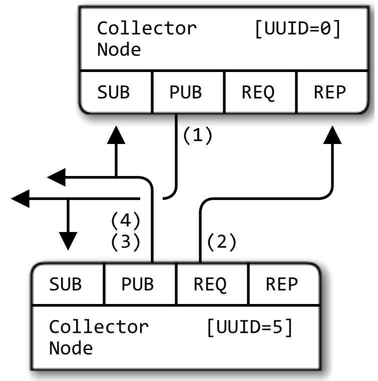


Figure 4.21: Root Recovery Protocol Diagram: (1) `WUTUP`, (2) `YO`, (3) `WUTUP`, (4) `IWIN`

The `WUTUP` and `IWIN` messages are shorthand for `TOPO(key="/RECOVERY/wutup"` and `TOPO(key="/RECOVERY/iwin"` respectively. The `YO` message is shorthand for `CONTROL(command="yo")`.

Chapter 5

Analysis

To verify *dCAMP* meets both the transparency and scalability distributed performance framework criterion outlined in Chapter 1, several experiments were run on an installation of *dCAMP* in a test environment. The goal of these experiments was two fold: measure *dCAMP* 's transparency in a real-world environment as well as determine the thresholds for several key configuration parameters as *dCAMP* scales.

Any DPF can be configured in such a way that it impacts the performance of the system being monitored, for example by collecting and reporting every available global metric and per-process metrics at the fastest sampling period. Therefore, it is necessary for the system administrator to know what reasonable configuration values should be used to monitor a given distributed system.

Likewise, for *dCAMP* to scale, it is important for the number of child nodes per parent to be limited to a reasonable number. These experiments help to define "reasonable" for various scenarios, environments, and performance monitoring requirements.

5.1 Transparency

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.

5.1.1 Workload

Apache JMeter[13] (v2.11) is used to run load against a default-configured Apache instance on a Lenovo Thinkpad (dual 2.16GHz Centrino Duo T2600, 2GB 667MHz DDR2, SATA) running Ubuntu 13.10. The client machine, a MacBook Pro (2.7GHz Core i7, 8GB 1333MHz DDR3, SSD) running OSX 10.9, is directly connected to the Apache server via crossover gigabit Ethernet.

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 (see Figure 5.1), and
3. download a 5MB file of random binary data.

The 25th Fibonacci workload is CPU-bound, and the 5MB download is disk-bound; the home page workload is only used to seed the client connection and is not part of the analysis measurements. 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.

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.

```
<?php
function F($n) {
    if ($n == 0) { return 0; }
    if ($n == 1) { return 1; }
    return F($n - 1) + F($n - 2);
}
?>
<html>
  <body>The 25th Fibonacci number is <?= F(25) ?>.</body>
</html>
```

Figure 5.1: Recursive 25th Fibonacci PHP Script: A naive approach was used in the implementation of `F()` in order to put more load on the server CPU.

5.1.2 *dCAMP* 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`). Below are the various sample periods used for the transparency test runs.

- **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

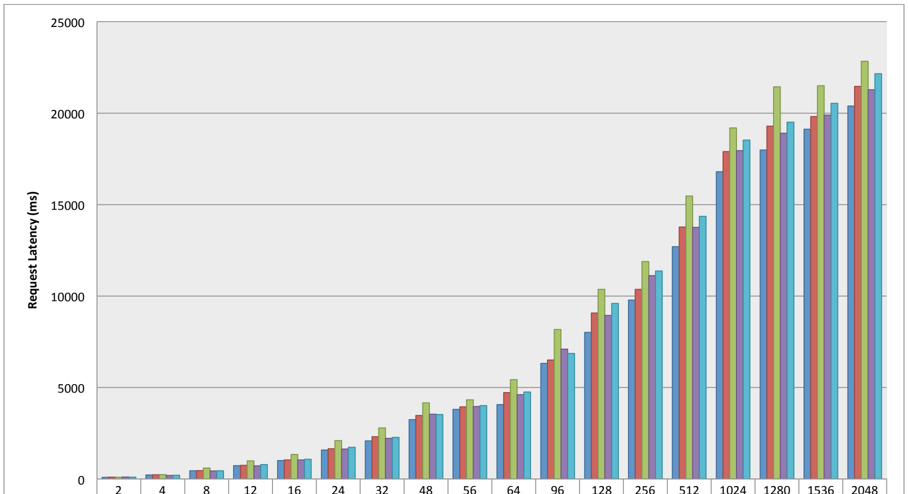
- **1s** – global metrics every 300 seconds, Apache metrics every 1 second, heartbeats every 300 seconds

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

5.1.3 Results

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. 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.

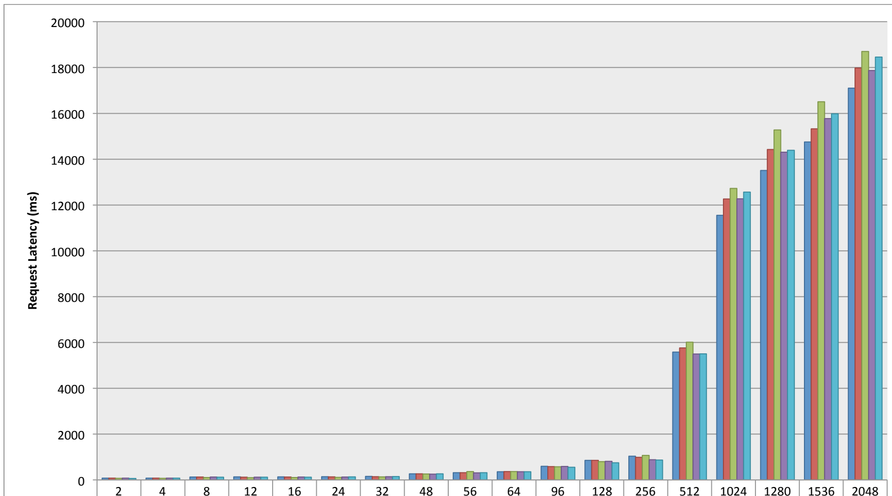
The rate at which request latency worsens begins to level off starting at the 512 thread load point. This is also the load point at which Apache begins to return errors. As the percentage of requests resulting in errors increases, the latency of the successful requests improves slightly. This explains the trend line shift.



	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

Figure 5.2: 25th Fibonacci Results

Apache's disk-bound performance measured in the 5MB download test is relatively unaffected by *dCAMP* . This 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.



	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

Figure 5.3: 5MB Download Results

A few observations can be drawn from these results.

When nodes are not expected to fail frequently, using longer heartbeat periods reduces the impact *dCAMP* has on the system. It is better to monitor a process using a faster sample period than an entire system using a slower sample period. The *dCAMP* system impact is noticeable but a considerably smaller factor than the impact hardware limitations have on performance monitoring.

5.2 Scalability

One of the primary measures of scalability for a distributed system is its network traffic.[20] By simulating successively larger *dCAMP* systems (with respect to node count), once can extrapolate *dCAMP* 's effectiveness at monitoring large distributed systems and how to best configure its metric collections.

5.2.1 Workload

dCAMP is setup to monitor a machine's global metrics, scaling the number of simulated nodes in the *dCAMP* system from three nodes (one Root, one Collector, one Sensor) up to 200 nodes (eight groups with twenty-five nodes per group). 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.

5.2.2 *dCAMP* 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

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

5.2.3 Results

Sparklines of each load point show the same pattern: highest network traffic occurs during start up and then also on shutdown. This pattern follows the design of *dCAMP* which uses a chatty configuration protocol and a terse data protocol. The rest of steady operation shows expected low network traffic except on sample periods.

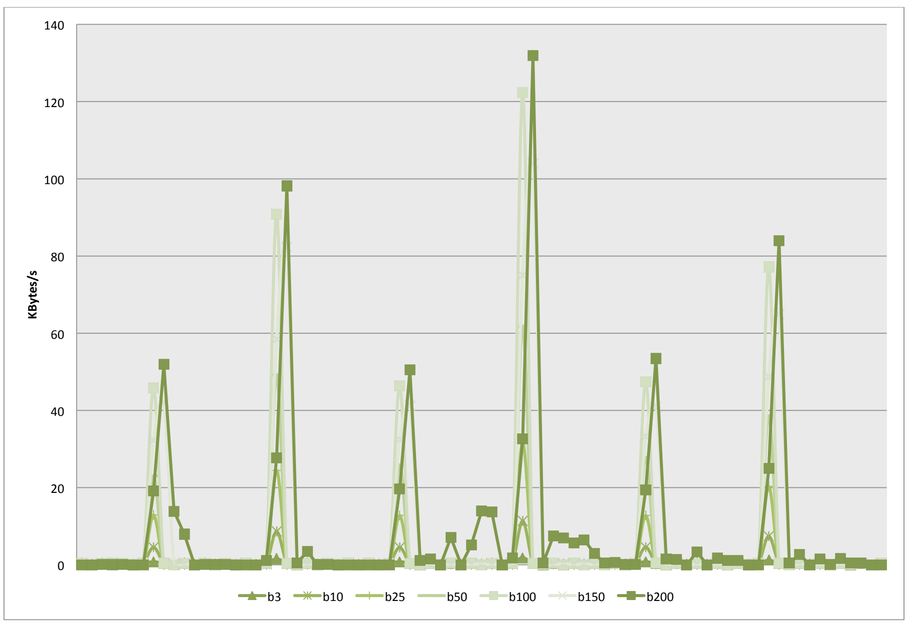


Figure 5.4: Network bytes during steady operation as the number of *dCAMP* nodes increases.

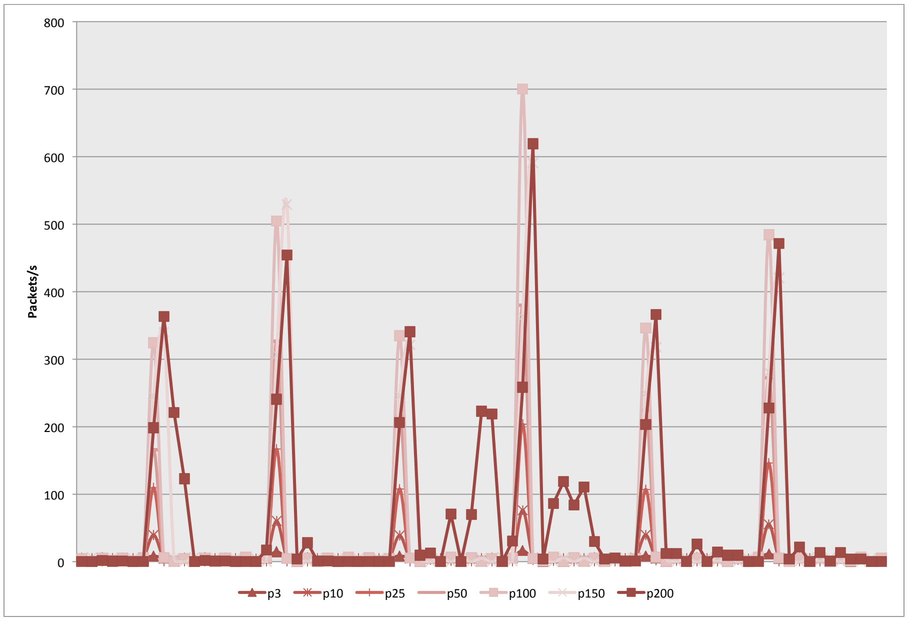


Figure 5.5: Network packets during steady operation as the number of *dCAMP* nodes increases.

As the node count increases, the rate at which bytes/packets are sent and received increases. This correlates with the larger configuration which *dCAMP* must track as well as the additional nodes sending and receiving data. Looking at the same values but also relating them to the number of nodes in the system, one sees the configuration size grows faster than the number of nodes.

However, the number of messages being sent per node actually goes down and levels off just under 1 packet per node per second. This can be attributed to the fact that the number of Sensor nodes increases faster in relation to the number of Collector nodes. 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. So as this ratio increases, it is expected the number of messages per node to decrease. This latter observation indicates a higher number of child nodes per parent would result in lower network utilization and better *dCAMP* scalability.

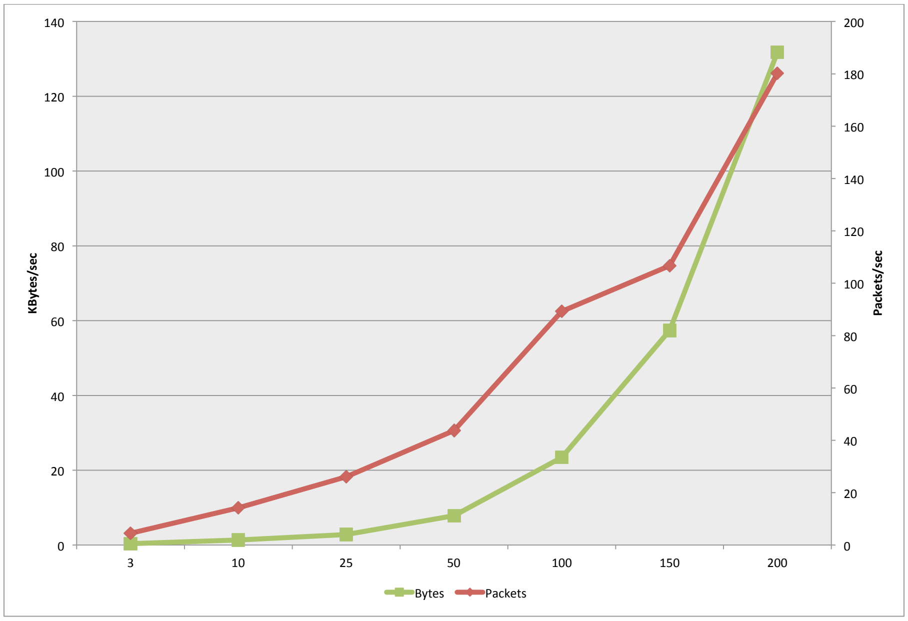


Figure 5.6: Average Bytes/Packets as the number of *dCAMP* nodes increases.

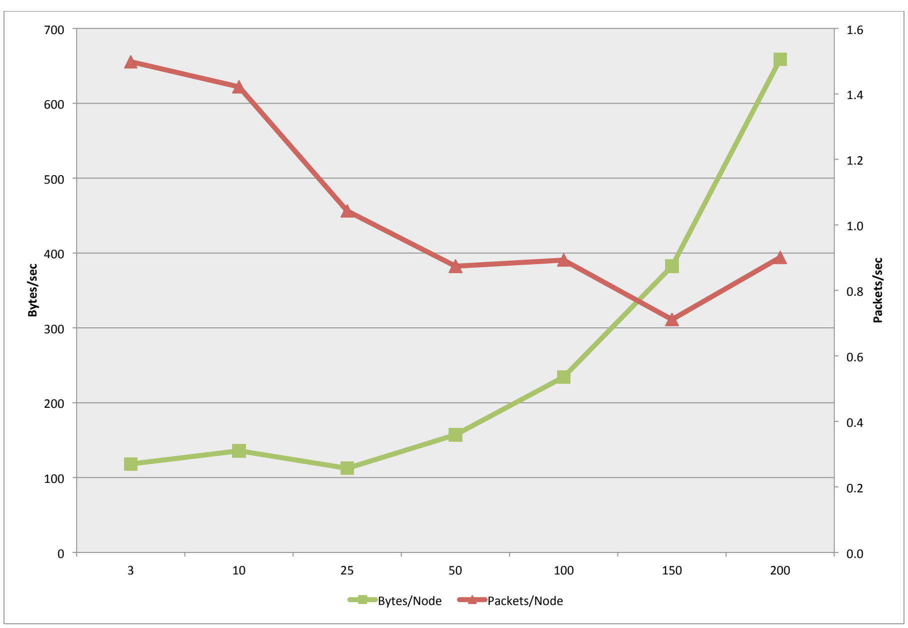


Figure 5.7: Average Bytes/Packets Per Node as the number of *dCAMP* nodes increases.

Chapter 6

Related Work

THIS SECTION IS FROM MY 508 PAPER

AND NEEDS TO BE CLEANED UP

Being distributed, a framework must collect data from a large number of nodes and aggregate the data to one node or client. Implementations have been built using centralized, hierarchical, peer-to-peer and any number of other architectures. There are 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.

There are a number of distributed performance frameworks being actively researched and developed, both academically and commercially. This work defines a set of criteria for evaluating distributed performance frameworks to measure the usefulness and viability of the framework. In the next section I list similar survey work, showing the novelty of this paper. Section 3 presents the evaluation criteria, section 4 provides the evaluation results of several distributed performance frameworks, and section 5 concludes.

6.1 Related Work

Several papers have been published dealing with this topic and which try to give an overview of the current state of distributed performance frameworks. This work, while related, differs in two main aspects: (1) the evaluation criteria presented here is more complete than previous works' requirements and (2) this work looks specifically at distributed performance frameworks rather than distributed system or grid tools and frameworks.

Zoraja et al. include a section on Monitoring Issues in their work on OMIS/OCM, CORAL, and MIMO [21] focusing on design and implementation issues for monitoring systems. These monitoring issues share some critical points with the criteria presented here, but the work lacks an analysis of other distributed performance frameworks and only includes limited analysis of the presented frameworks against the listed monitoring issues. [12] provides a short survey of related monitoring systems in comparison to the ZM4/SIMPLE monitoring environment. While this survey does look at a number of important framework characteristics, it is restricted to hybrid distributed performance frameworks.

Allan and Ashworth published an exhaustive survey of distributed computing tools [2] but did not present any particular focus on distributed performance frameworks nor did they include a set of criteria for evaluating frameworks. Zanikolas et al. present an in-depth taxonomy of grid monitoring systems [20], including a set of general requirements for monitoring systems. Their work does much for defining a good set of criteria for evaluating and analyzing distributed performance frameworks, but has a much broader scope than this work. This work expands on their monitoring system requirements to account for the completeness and validity of distributed performance frameworks.

6.2 Analysis Results

The distributed performance frameworks analyzed in this section were chosen based on their categorization in the [20] taxonomy; only level 2 frameworks are included. Level 2 frameworks are defined as having at least one type of republisher in addition to producers; these frameworks usually distribute functionality across multiple hosts. [20] 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.

6.2.1 NetLogger

Work done by Brian Tierney and Dan Gunter [18] [7] presents the Network Application Logger Toolkit (NetLogger). This framework can be used to monitor the performance of distributed systems at a very detailed level. With a new logging format and activation service [8] the authors improved upon their previous work and increased the toolkit's scalability and data delivery models. NetLogger is being actively developed and is one of the more well known distributed performance frameworks. The toolkit is composed of four parts: an API and library for instrumenting a given application, a set of tools for collecting and sorting logs, performance sensors, and a visualization user interface for the log files.

Each part assumes the system clocks of the individual nodes are accurate and synchronized (the authors mention the use of NTP to achieve a required clock synchronization of one millisecond). The instrumentation of code allows NetLogger to gather more detailed data from an application-to-application communication path, such as traces of network packets through a call hierarchy. The instrumentation also allows the activation service to update the monitoring of parts of the system dynamically as consumers subscribe to various events and metrics.

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.

6.2.2 JAMM

Java Agents for Monitoring and Management (JAMM) [17] is the fruit of work by the authors of NetLogger to build a monitoring system with managed sensors.

The JAMM system consists of six components: sensors, sensor managers, event gateways, directory service, event consumers, and event archives. There is a sensor manager on each host, with the sensors acting as producers for the gateways which they publish the data to. The gateways can then filter and aggregate the incoming data according to consumer queries. The directory service is used to publish the location of the sensors and gateways, allowing for dynamic discovery of active sensors by the consumers. The event archive is used for historical analysis purposes.

JAMM explicitly uses a pull data delivery model where data is only sent when requested by a consumer. The overall architecture is generally distributed with the directory service being centralized. 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.

6.2.3 Hawkeye

Hawkeye [9] is a monitoring and management tool for distributed systems which makes use of technology previously researched and developed as part of the Condor project [14]. Condor provides mechanisms for collecting information about large distributed computer systems. Hawkeye is being readily developed and is freely available for download on Linux and Solaris.

Hawkeye uses a general push delivery model by configuring Condor to execute programs, or modules, at given time intervals, collect performance data, and send it to the central manager. These modules are configurable such that the "period" of module execution can be set to a given time frame in seconds, minutes, or hours or the module can be executed in "continuous" mode where the module's execution never ends. The available modules for monitoring a Condor pool include: disk space, memory used, network errors, open files, CPU monitoring, system load, users, Condor Node, Condor Pool, and Grid Probe. Custom modules can also be developed and installed for monitoring of arbitrary resources and metrics. Data can be accessed from the central manager via an API, CLI, or GUI.

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.

6.2.4 SCALEA-G

Truong and Fahringer present SCALEA-G [19], an unified monitoring and performance analysis system for distributed systems. It is based on the Open Grid Service Architecture [4] and allows for a number of services to monitor both grid resources and grid application. SCALEA-G uses dynamic instrumentation to profile and trace Java and C/C++ applications in both push and pull data delivery models, making the framework both scalable and portable.

The SCALEA-G framework is composed of several services: directory service, archival service, sensor manager service, instrumentation service, client service, and user portal. These services provide the following functionality respectively: publishing and searching of producers and consumers, storage of performance results, management of sensors, dynamic instrumentation of source code, administering clients and analyzing data, and on-line monitoring and performance analysis.

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.

6.2.5 IMPuLSE

Integrated Monitoring and Profiling for Large Scale Environments [3] was designed to address "operating system-induced performance anomalies" and provide "accurate, low-overhead, whole-system monitoring." The authors have chosen to develop a message-centric approach which associates data with messages rather than hosts and a system-wide statistical sampling to increases the framework's scalability.

The IMPuLSE framework is still in the design stage, and therefore lacks any implementation data outside of their new message-centric design pattern which shows promising results. Unfortunately, this leaves the framework with unknown transparency, security, completeness, portability, and validity.

6.3 Conclusion

There are a number of high quality and effective distributed performance frameworks being actively researched and developed, but with some frameworks having more research than others, there is a natural disparity of information about each framework. While the frameworks vary in distributed architecture and features, they all fulfill the minimum requirements of performance frameworks. The frameworks analyzed in this work are mainly software based sensor frameworks. This was chosen due to the inherent portability advantage of software sensors over hardware or hybrid sensors.

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. I leave it as future work to conduct formal experiments to test validity, transparency, and scalability of the distributed performance frameworks analyzed here.

6.4 NEW RESEARCH

- http://host-sflow.sourceforge.net/about.php

  "The Host sFlow agent exports physical and virtual server performance metrics using the sFlow protocol. The agent provides scalable, multi-vendor, multi-OS performance monitoring with minimal impact on the systems being monitored."

  Also capable of application layer monitoring (e.g. Apache, Tomcat, node.js, Memcached).

- http://ganglia.sourceforge.net/ (http://ganglia.info/papers/science.pdf)

  "Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and Grids. It is based on a hierarchical design targeted at federations of clusters. It leverages widely used technologies such as XML for data representation, XDR for compact, portable data transport, and RRDtool for data storage and visualization. It uses carefully engineered data structures and algorithms to achieve very low per-node overheads and high concurrency."

- https://pypi.python.org/pypi/nvidia-ml-py/

  Python Bindings for the NVIDIA Management Library; provides access to GPU performance metrics.

Chapter 7

Future Work

Additional Features

These features would provide a more complete, end-to-end solution for distributed performance monitoring.

- end-to-end tool built on top of *dCAMP* ; aggregation provides benefit of looking at large parts of system quickly; you can look at the aggregate and drill down to find the problem nodes
- Lightweight web server on each node to support REST APIs and graphical interface.
- security — send salted passphrase with every control message, encrypt all messages, etc.
- provide full support for metric sampling in collector role (instead of requiring two nodes on single machine)

Fault Tolerance

Improve the fault tolerance of *dCAMP* by implementing these features which were out-of-scope for the original project.

- network failure: *dCAMP* does not support any fault tolerance for network failures; *dCAMP* only attempts to recover from node failures. It is assumed that if (part of) the network goes down, the lack of data from that subnet will suffice. specifically, *dCAMP* does not tolerate the split brain syndrome[1].
- time accuracy: The system time among multiple nodes in the system may vary significantly; *dCAMP* is not meant to be a high-resolution system with respect to the order of performance data occurrences. It is assumed that NTP provides sufficient time synchronization across all nodes in the system OR the precise ordering of performance events in the system is not required.
- self-restarting executable for base node
- detect collector failure in the root role, not just metric role
- Root node does not need to be present for the system to operate. 1) Root (i.e. 'Management' service) node can come and go, 2) all data is stored at top-level Collector nodes, 3) Root comes online and asks Collectors for data (since T?), 4) there is a "master" Collector node for config (is that same as root?)

Improve Performance and Scalability

Increase efficiency of protocols and make really large *dCAMP* systems possible.

- use IOLoop (green events?) instead of polling sockets for messages
  - look into using an io loop with callbacks to better manage sockets
  - possibly, use a single io loop, hosted by the base node service and passed to the roles it starts?
  - this could greatly increase the scalability of dCAMP and reduce its overhead
  - except the sensor service, most threads sit idly waiting for messages to arrive; this is a waste of system resources; using events reduces complexity of implementation and improves runtime performance
- multicast topo discovery – reduce config cost, use gateways to multicast into other subnets
- multiple-level branches are not supported in the current implementation. that is, all collector nodes have the root node as their parent and only have leaf nodes as their children.
  - support would allow large group configurations to be automatically split into multiple (identically configured) branches for improved scalability

Metric Extensions

Only a small subset of metrics were implemented in *dCAMP* as a proof of concept. The rest of the full set listed in *dCAMP* Metrics section are left as future work.

VARIABLE LENGTH DATA Does *dCAMP* need to support arbitrary data lengths?

NESTED METRICS It could be possible for data messages to contain nested data messages, e.g. average/sum of rates.

GROUPINGS *dCAMP* may need a more compact data message format for combining multiple metrics into a single message, e.g. for aggregation purposes or representing entire branches in the topology. group by source/type/time? group start/end frames?

HISTOGRAMS *dCAMP* should provide a metric histogram ability. Perhaps this should done at each node or only at the root.

METRIC REQUESTS The admin can use *dCAMP* to request metrics on a one-time basis, for example, to enumerate the available disks on each node. The key here is a metric collection is not based on configuration file but rather on real-time input from the end-user.

Perhaps there should be some special, e.g. “once”, metric collection specifications so config data can be sent up to the root only at node start.

Chapter 8

Conclusions

Bibliography

[1] Reference needed.

[2] R. Allan and M. Ashworth. A survey of distributed computing, computational grid, meta-computing and network information tools, 2001.

[3] P. G. Bridges and A. B. MacCabe. Impulse: integrated monitoring and profiling for large-scale environments. In *LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–5, New York, NY, USA, 2004. ACM.

[4] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6):37–46, 2002.

[5] M. Gabel and M. Haungs. Camp: a common api for measuring performance. In *LISA'07: Proceedings of the 21st conference on Large Installation System Administration Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.

[6] Ganglia. http://ganglia.info/.

[7] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. Netlogger: A toolkit for distributed system performance analysis. *International Sympo-*

sium on Modeling, Analysis, and Simulation of Computer Systems, 0:267, 2000.

[8] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer. Dynamic monitoring of high-performance distributed applications. In *Applications, Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing*, pages 163–170, 2002.

[9] Hawkeye. http://www.cs.wisc.edu/condor/hawkeye/.

[10] P. Hintjens. Clustered hashmap protocol. http://rfc.zeromq.org/spec:12/CHP, April 2011.

[11] P. Hintjens. *Code Connected Volume 1, Learning ZeroMQ*. CreateSpace Independent Publishing Platform, January 2013. http://zguide.zeromq.org.

[12] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle. Distributed performance monitoring: Methods, tools and applications. *IEEE Transactions on Parallel and Distributed Systems*, 5:585–597, 1994.

[13] Apache jmeter. http://jmeter.apache.org/.

[14] M. Litzkow, M. Livny, and M. Mutka. Condor-a hunter of idle workstations. *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, Jun 1988.

[15] R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, Jun 1997.

[16] M. Sustrik, M. Lucina, and P. Hintjens. zmq_socket - create ømq socket. http://api.zeromq.org/3-2:zmq-socket, 2012.

[17] B. Tierney, B. Crowley, D. Gunter, J. Lee, and M. Thompson. A monitoring sensor management system for grid environments. In *Proceedings of the IEEE High Performance Distributed Computing Conference (HPDC-9)*, pages 97–104, 2000.

[18] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The netlogger methodology for high performance distributed systems performance analysis. In *In Proc. 7th IEEE Symp. on High Performance Distributed Computing*, pages 260–267, 1998.

[19] H.-L. Truongc and T. Fahringer. Scalea-g: A unified monitoring and performance analysis system for the grid. *Sci. Program.*, 12(4):225–237, 2004.

[20] S. Zanikolas and R. Sakellariou. A taxonomy of grid monitoring systems. *Future Gener. Comput. Syst.*, 21(1):163–188, 2005.

[21] I. Zoraja, G. U. Rackl, and T. Ludwig. Towards monitoring in parallel and distributed environments. In *University of Split*, pages 133–141, 1999.

Appendix A

ZeroMQ Primer

A.1 Why ZeroMQ

Not surprisingly, the most succinct description of ZeroMQ is found in *The Guide*[11] preface,

> 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.

No appendix could justly explain ZeroMQ or give the reader a true understanding of its abilities and proper use. Read *The Guide* (a freer and more up-to-date version is online at http://zguide.zeromq.org) and, if truly adventurous (or just morbidly curious), go through all 750+ examples in any of the 28 programming languages available.

Forget about RPC, MPI, and raw sockets. ZeroMQ allows a developer to build distributed systems by focusing on the data and implementing simple design patterns. In short, ZeroMQ allows the distributed systems developer to have fun. No joke.

A.2 Sockets and Message Patterns

To begin understanding ZeroMQ, a foundational knowledge of ØMQ sockets, messages, and patterns is needed.

A.2.1 Sockets and Messages

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.

First, they are asynchronous—the actual sending and receiving of data on a ZeroMQ socket is handled by a background thread. Second, ØMQ sockets have built-in support for one-to-many connections. That is, a single socket can send and receive data from multiple endpoints.

ZeroMQ sockets are explicitly typed, with the type dictating how data is routed and queued to and from the socket. Furthermore, this explicit typing means only certain socket types can be connected to each other.

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.

A.2.2 Messaging Patterns

Generally speaking, the ZeroMQ messaging patterns are defined by the socket routing and queuing rules as well as each socket's compatible type pairings. As listed in the `zmq_socket` man page[16], ZeroMQ supports the following core 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."

The two socket types used for this pattern are `PUB`, which can only send messages, and `SUB`, which can only receive messages. Naturally, this is a unidirectional pattern.

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."

The two basic socket types for this pattern, `REQ` and `REP`, require strict ordering of messages: a message must be first be sent on the `REQ` socket before a message can be received on the socket, and vice versa for the `REP` socket. Two advanced socket types, `XREQ` (or `DEALER`) and `XREP` (or `ROUTER`), allow a more lenient communication pattern.

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."

PUSH (send-only) and PULL (receive-only) socket types are used for this pattern. Like Pub-Sub, this is a unidirectional pattern.

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."

Only the PAIR socket type can be used for this pattern.

A.3 Useful Features for *dCAMP*

Apart from the general happiness ZeroMQ offers to the distributed systems developer, some features are particularly useful in *dCAMP* .

A.3.1 Topic Filtering

All messages sent using Pub-Sub are filtered (usually by the publisher) based on the "topics" to which each subscriber subscribes. Topics, stated simply, are the leading bytes of a message's first frame. By default, a SUB socket is not subscribed to any topics.

Consider a PUB message which contains `b'/fruit/apple'` as its first frame. A subscriber would receive this message if it subscribed to `b'/fruit'` or `b'/'` or even `b''` (an empty frame). But if the subscriber was not subscribed to any topics or only subscribed to the `b'/plants'` topic, it would not receive the message. Do note: the topic can be any binary data, not just character data.

Topic filtering fits naturally into the Configuration Replication Protocol where different roles only replicate portions of the config and the Topology Discovery Protocol where the root node needs to send commands to a subset of the topology.

A.3.2 Easy Message Debugging

ZeroMQ's atomic multipart message passing lends itself to what *The Guide* calls the "Cheap or Nasty pattern" [11]. That is, use cheap, easy to write/read, overly verbose messages for infrequent control scenarios and nasty, compact, highly performant messages for long-lived and frequent data scenarios.

In *dCAMP* , all control messages follow the cheap pattern, making them easy to debug. But the data messages, which tend to not need a lot of debugging, are free to be more optimized.

A.3.3 Simplified Threading Design

While much care must still be taken in their use, the inherent properties of ZeroMQ sockets (asynchronous nature, utilization of send/receive queues, ability to round-robin and fair-queue messages) allow for more attention to be paid to the design, purpose, and real work of each thread rather than the mechanics of sending and receiving data.

This is clearly seen in *dCAMP* Service implementations where, for example, a single thread can be used to process both remote nodes' performance metrics and the local node's performance metrics without any special coding: one socket with multiple endpoints connected and the built-in fair-queuing taking care of routing.

A.3.4 Quick Simulation

The nature of ZMQ bind/connect endpoints just being a transport and a network address means simulation of large distributed systems can take place on a single machine using unique port numbers. Additionally, the interoperability of `inproc` (within the same process) and TCP transports allows for even larger simulations, not bound by port availability on the host.

This is demonstrated several times in examples within *The Guide*, and it indeed held true during *dCAMP* development and testing.

Appendix B

Real Life

Finishing a master's thesis after college is no joke. Get it done now. Do not waste your time. Real life is full of real work. Marriage. Babies.

Life is not waiting for you to complete unfinished tasks. There will be a tinge of guilt with every new project and every day spent not working towards completion of the thesis. Goodwill runs out; good intentions are just false promises.

Also, a degree is a nice thing to have, if not now, at least at some point in the future.