

Network Working Group
Internet-Draft
Intended status: Informational
Expires: April 24, 2014

S. Whyte
Google Inc.
M. Hines
W. Kumari
Google, Inc.
October 21, 2013

Bulk Network Data Collection System
draft-swhyte-i2rs-data-collection-system-00

Abstract

Collecting large amounts of data from network infrastructure devices has never been very easy. Existing methods generate CPU and memory loads that may be unacceptable, the output varies across implementations and can be difficult to parse, and these methods are often difficult to scale. I2RS programmatic interfacing with the routing system may exacerbate this problem: state needs to be collected from nodes and fed to consumers participating in the control plane that may not be physically close to the nodes. This state includes not only control plane information, but elements of the data plane that have a direct impact on control plane behavior, like traffic engineering.

This document outlines a set of use cases requiring a flexible framework to collect routing system data, and the features and functionality needed to make such a framework useful for these use cases.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119](#) [[RFC2119](#)].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any

Internet-Draft

Bulk Data Collection

October 2013

time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 24, 2014.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1.	Introduction	3
2.	Desired functionality	3
2.1.	Database Model	4
2.2.	Pub-Sub	4
2.3.	Capability Negotiation	5
2.4.	Format Agnostic	5
2.5.	Transport Options	5
2.6.	Filtering	5
2.7.	Timestamps	6
2.8.	Introspection	6
2.9.	Registration	6
3.	Use cases	6
3.1.	Push	7
3.1.1.	Interface counters	7
3.1.2.	Thresholds	7
3.1.3.	Streaming	7
3.2.	Pull	7
3.2.1.	Interface counters	8
3.2.2.	RIB Dump	8
3.2.3.	Arbitrary data collection	8
3.3.	Dynamic subscriptions	8

4.	Subscriber versus consumer	8
4.1.	Remapping	8
5.	Errors	9
6.	IANA Considerations	9
7.	Security Considerations	9

8.	Acknowledgements	9
9.	References	10
9.1.	Normative References	10
9.2.	Informative References	10
	Authors' Addresses	10

[1.](#) Introduction

Managing and monitoring a network requires getting state out of it. You can't manage what you don't measure, as the saying goes. Currently there are a limited set of tools to get data off of network nodes, and they do not lend themselves to programmatic access.

The primary tool today is SNMP. SNMP can be used to both push data off a node (via traps/notifications) and pull data off the box (via queries). SNMP queries have a variety of issues, not the least of which is the fact that the protocol specification requires data structures to be created on demand on network nodes that do not match how the device's operating system data structures store the same data. Fixing this problem has the immediate benefit of reducing CPU and memory consumption of the monitored network devices, greatly increasing the deployability and relevance of a solution. SNMP traps /notifications suffer from a lack of introspection; the network management system (NMS) must be preconfigured to understand what information is being reported.

Other tools include CLI scraping and Syslog. CLI scraping is a low-level pull mechanism and essentially the opposite of programmatic access. Any change in CLI implementation, whether its a simple whitespace correction, re-ordering of configuration stanzas, typographical errors, or even unit changes, can require a rewriting of monitoring software. This is compounded by the fact there is no standardized CLI specification, such that a network with multiple vendors in it requires these rewrites per vendor CLI change.

Syslog is another way to push data off of a network node. Syslog has

been around a long time, and while current standards provide structured data output, very few implementations exist on network nodes currently. For the most part NMSes must be trained how to consume and interpret different implementations of syslog.

[2.](#) Desired functionality

Collecting large data sets with high frequency and resolution, with minimal impact to a device's CPU and memory, is the primary objective. Aspects of the over-all data collection system, such as availability or reliability or scaling, are outside of scope as they deal with the data once it has left the network node.

Whyte, et al.

Expires April 24, 2014

[Page 3]

Internet-Draft

Bulk Data Collection

October 2013

We are only focusing on getting data off the node in an easily machine parsable format.

[2.1.](#) Database Model

A database model is desired, whereby a network node can describe the data it has available, and the structure of that data. This gives the implementor the ability to present a database model that can be optimal with the node's internal data structure implementations. The NMS consumes and understands the database model only after it has been trained to do so by incorporating a published version of the database model from the vendor.

It should be noted that all existing data collection methods outlined earlier require explicit knowledge of the method's implementation for integration into a NMS. We do not propose a solution that eliminates this, because heterogeneity of the data is not required, as we can see from existing implementations. Rather, capability negotiation and flexible formats and transports, outlined below, are desired enabling the primary objective of getting large data sets off the nodes with as little impact as possible.

[2.2.](#) Pub-Sub

An underlying pub-sub model is desired for a variety of features. It provides a security model for authorization, it supports intermediaries allowing the system to scale as needed, and it provides both push and pull methods of data distribution.

In the context of this draft, a pub-sub model is a general concept indicating information flow. Specific system details are obviously critical yet belong in a data model document. The high level desire is to have network nodes as publishers, with an NMS implementing subscribers. Conceptually, they are connected by a message bus, a layer of indirection between the publishers and subscribers. Having a message bus allows publisher fan-in, subscriber fan-out, and a number of other useful features outside the scope of this document. The message bus is frequently referred to as a broker inside pub-sub models.

Having a message bus abstraction allows for considerable flexibility in NMS design as well. Placement of brokers in the network, their redundancy, availability, scaling per publisher or subscriber, can all be tailored to suit an individual network's needs, from extremely simple (flat) to extremely complex with multiple layers of hierarchy. Many implementations of pub-sub models exist, scaling both in number of subscriptions and in number of messages, both of which should be considered carefully in the I2RS context.

[2.3.](#) Capability Negotiation

Capability negotiation allows a node to inform a subscriber of a number of options. Two extremely important options would be transport protocols and formats supported. Other aspects such as security options and error handling would also be negotiated during this phase.

The capability negotiation phase is done via a control channel opened for the purpose of registering subscriptions with the node. This control channel should be TCP.

[2.4.](#) Format Agnostic

From the I2RS perspective, this framework should be format agnostic. If a node advertises the ability to present data in XML and the subscriber agrees, then XML can be used. Other formats that have interest are JSON, HTML, and protobufs. Even interest for /proc/net formatted output exists, and would help a NMS based on this framework integrate into existing server configuration management systems.

[Editor note: even ASN.1 should be an acceptable format. This would

potentially allow an extremely easy deployment into an existing SNMP based NMS.]

[2.5.](#) Transport Options

Because the focus of this framework ends at getting data off the box as quickly as possible, implementations should have the freedom to choose a transport that meets their system design needs and not be restricted by a specific format.

During the negotiation phase a node should advertise all the transport options it provides and allow the subscriber to select what it needs.

Given the time-value of different data elements coming off the node can be quite different, it should be possible to request multiple transports and associate a subscription with the transport protocol of choice.

[2.6.](#) Filtering

Once a network node has provided its database model to a subscriber, the subscriber needs a way to select parts of the model for subscription, and it needs to be able to request multiple subscriptions at a time.

This framework should provide a standard filtering mechanism so that, independent of the database model structure and contents, a subscriber can select interesting items to collect and bucket them based on standard parameters such as frequency of collection, underlying transport required, whether the data is to be pushed or pulled, or even streaming or one-shot.

[2.7.](#) Timestamps

Every piece of data collected by this framework needs a timestamp associated with it indicating when the node made it available for collection. This is not required on a per-variable basis, for example data organized into a table only requires a timestamp associated with the table.

This is not to say additional timestamps are not useful for certain data sets nor that other timestamps with other semantics, for example collection time versus advertisement time, can not be used, but rather those additional timestamps are better placed in the database model supported by the device.

[2.8.](#) Introspection

This framework should support introspection of the database model. Introspection provides support for data verification, easier inclusion of legacy data, and easier merging of data stream.

[2.9.](#) Registration

After capabilities and a database model have been exchanged, and a filter used to select elements of the model to subscribe to, the framework should support a standard way to register for all the data desired, using whatever capabilities were advertised by the node.

Once registration is complete, the control channel can be closed. Ensuring subscriptions are correct, complete, and replicated or not, is up to the overall system and not the network node.

[3.](#) Use cases

Following are example use cases outlining the utility of subscribing to data with different parameters.

[3.1.](#) Push

Pushing data off the box can be done synchronously at fixed intervals, or asynchronously in an ad-hoc fashion. All data pushed is set up via registered subscriptions.

[3.1.1.](#) Interface counters

Interface counters provide a use case demonstrating the need to push data off of a network node at specific intervals. In this proposed framework, a node would advertise its database model including all the interfaces it has to offer and what it can count on each. A subscriber would select the interfaces and counters of each it is interested in via a filter, use the filter to group them according to available parameters, and register with the node to have them published at agreed upon intervals.

[3.1.2.](#) Thresholds

Another use case demonstrating a push capability is thresholding. Assuming a node advertises the capability to record and track a threshold for a particular data type, it would use the registered subscription to push relevant data to the subscriber whenever the threshold was crossed. As an example, a subscriber may want to set a threshold for memory consumed – if the available device memory falls below a threshold the subscriber should be informed so that the operator can investigate the issue manually or programmatically.

[3.1.3.](#) Streaming

Streaming data, such as RIB information, will be critical to supporting I2RS functionality. In this use case, a subscriber may desire to have all updates to a RIB streamed into the collection system, in as close to real-time as possible.

[3.2.](#) Pull

Pulling data off the node will always be a one-shot function. As such it is probably the most heavy-handed way to get data into the collection system, as it requires all the overhead of setting up and tearing down the control channel, exchanging the database model, creating a filter, and receiving the data. Nevertheless, it can be a valuable option and should be supported.

n.b. it is certainly possible to cache requests on publishers, and have them "replayed" via a subscription identifier. However the capability to track the state required to do so may not be available on a node, and this is somewhat counter to the overall goal of

minimizing impact to the node. Having this capability as an optional

parameter of a database model, is worth exploring.

[3.2.1.](#) Interface counters

Similar to the interface counter example above, except in this case the registration includes a parameter indicating the data should be collected immediately and sent only once.

[3.2.2.](#) RIB Dump

Getting a snapshot of the node's current RIB can be useful for a variety of reasons. Similar to collecting RIB information above, in this example the subscriber would register for a one-shot dump of the RIB, collected and sent immediately.

[3.2.3.](#) Arbitrary data collection

Once the NMS understands a node's database model, it should be able to register for one-shot collection of any subset of that database model. Given the overheads involved, this would best be restricted to one-off collection needs, such as troubleshooting, but the use case need is solid.

[3.3.](#) Dynamic subscriptions

This framework should support dynamic subscription capabilities with pre-existing monitoring protocols that currently require static configuration. For example, if a node's database model indicates it support IPFIX, using the standard registration process outlined above a subscriber should be able to set up a streaming IPFIX feed. BMP and the like should also be available via this mechanism.

[4.](#) Subscriber versus consumer

It should be noted that because overall data collection system architecture is out of scope, it is opaque to this framework whether a subscriber is also the consumer of data. In order to maximize design options, including scalability of the overall system, both options should be supported.

[4.1.](#) Remapping

Remapping in this context is the ability to modify a node's database model and request the modified model be used in subscriptions. While this has interesting properties, it strays far from the primary objective of getting data off of nodes as fast as possible with as little impact as possible, and thus should be considered out of scope.

[5.](#) Errors

Errors happen. Many classes of errors and their handling are already well-understood and don't need to be re-iterated here. There are certainly failure modes that may be unique to I2RS or this framework, however, and we should be prepared to incorporate solutions for those.

For example, providing a method for a node and a subscriber to agree on resolution steps after defined error events would be very useful. A subscriber may want certain subscriptions to be available for pulling, if the push mechanism failed.

There may also be value in defining how a subscriber can probe the transport layer, such that publisher responses can assist in troubleshooting protocol-specific failures.

The framework needs to support standardized handling of stale data. This class of error will largely be related to handling changes and exceptions in the database models exchanged. For example what happens when a node's physical configuration changes and part of an existing subscription becomes invalid. Similar thought to logical changes, such as the disappearance of a BGP speaker, needs to be given.

[6.](#) IANA Considerations

This document makes no request of the IANA.

[7.](#) Security Considerations

I2RS provides security requirements, any security requirements raised by this framework should be encompassed there.

[TODO(WK, SW): This section needs more work / text]

[8.](#) Acknowledgements

The author wishes to acknowledge the contributions of a number of folk, including

{TODO(WK, SW): Remember to add folk! }

Internet-Draft

Bulk Data Collection

October 2013

[9.](#) References

[9.1.](#) Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), March 1997.

[9.2.](#) Informative References

[DeBoer] De Boer, M. and J. Bosma, "Discovering Path MTU black holes on the Internet using RIPE Atlas", July 2012, <<http://www.nlnetlabs.nl/downloads/publications/pmtu-black-holes-msc-thesis.pdf>>.

Authors' Addresses

Scott Whyte
Google Inc.
1600 Amphitheatre Parkway
Mountain view, California 94043
USA

Email: swhyte@google.com

Marcus Hines
Google, Inc.
1600 Amphitheatre Parkway
Mountain view, California 94043
USA

Email: hines@google.com

Warren Kumari
Google, Inc.
1600 Amphitheatre Parkway
Mountain view, California 94043
USA

Email: warren@kumari.net

Whyte, et al.

Expires April 24, 2014

[Page 10]