Technical Disclosure Commons

Defensive Publications Series

July 2020

APP-CENTRIC DISTRIBUTED MESSAGING BUS FOR PERFORMANCE MONITORING

Ted Hulick

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation
Hulick, Ted, "APP-CENTRIC DISTRIBUTED MESSAGING BUS FOR PERFORMANCE MONITORING", Technical Disclosure Commons, (July 22, 2020)
https://www.tdcommons.org/dpubs_series/3452

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.

APP-CENTRIC DISTRIBUTED MESSAGING BUS FOR PERFORMANCE MONITORING

AUTHOR:
Ted Hulick

ABSTRACT

A novel messaging system simplifies integration of dissimilar devices and instrumentation to enable a "distributed" application performance intelligence.

DETAILED DESCRIPTION

There are numerous viewpoints for instrumentation and monitoring of application flow and performance. Some current examples include:

- Application: application level - code level instrumentation, e.g., Application Performance Management (APM);
- Endpoint: OS level - mainly kernel level instrumentation, e.g., Machine Agent;
- Network: wire level - mainly layers 1-7 of the protocol stack, e.g., Network Performance Management (NPM);
- Container: specific to container/mainly peer to peer communication, e.g., Docker/Kubernetes; and
- Browser: user level - mainly javascript instrumentation, e.g., EUM/BRUM.

Further, there are many Network Infrastructure Modules (NIMs) involved in the application flow that have different levels of visibility into performance and availability. Vendors have long tried to find ways to leverage all of these instrumentation points by integrating them together. In most cases, the correlation is done on a backend processing system versus in real time. However, accomplishing this integration has proven to be more difficult than originally imagined due to:

- the instrumentation points being used by different groups (e.g., application stakeholders, network engineers, IT administrators, etc.);
- the instrumentation points not sharing or correlating information;
- the instrumentation points not being aware of each other; and
- the instrumentation points coming from different vendors.

For all of these reasons, no standard communication system exists that is convenient to all of these different viewpoints.

The described messaging system ties together these two distinct area in a "full duplex" membership fashion. The information can be used by other infrastructure networking modules such as SD-WAN, SDN, load balancers, routers, and switches either to display correlated information or makes decisions related to Quality of Service (QOS).

APM and NPM

APM (Application Performance Management) has been around for over 30 years, with top APM vendors including AppDynamics, New Relic, and Dynatrace. NPM (Network Performance Management) has been around for even longer and recently has evolved in many ways such as adding QOS control, analyzing traffic patterns, assessing transaction time, etc. Generally, NPM tools are "passive monitors" that plug into the "SPAN or Mirror ports" switches/routers/load balancers.

Each technology has a different "viewpoint (instrumentation)" in terms of how it measures performance. For the most part, APM instrumentation is embedded into the Application itself using runtime APIs and dynamic interception of key "points" in the transaction and taking measurements at those points. In the case of NPM, it is done at much lower level and based on a "packet by packet" basis. In the case of lower network layers, this would be the IP/TCP level looking at things like transmission and ack deltas, DNS requests/responses, network resets (TCP RST flag). Some NPM tools will measure higher in the stack at the HTTP level and even decoding Simple Object Access Protocol (SOAP) XML messages (although Representational State Transfer (REST) has replaced SOAP in most application stacks).

Network Infrastructure Module (NIM) Types

Todays networks are very diverse and contain many different types of Network Infrastructure modules, which are generally virtual or physical in nature. All of these modules can benefit by having access to APM metrics, alerts, and snapshots.

These modules include:

- SD-WAN – an acronym for software-defined networking in a wide area network (WAN). SD-WAN simplifies the management and operation of a WAN by decoupling the networking hardware from its control mechanism. This concept is similar to how software-defined networking implements virtualization technology to improve data center management and operation. A key application of SD-WAN is to allow companies to build higher-performance WANs using lower-cost and commercially available Internet access, enabling businesses to partially or wholly replace more expensive private WAN connection technologies such as Multiprotocol Label Switching (MPLS).
- SDN – Software Defined Networking (SDN) architectures decouple network control and forwarding functions, enabling the network control to become directly programmable and the underlying infrastructure to be abstracted from applications and network services. SDN Applications are programs that explicitly, directly, and programmatically communicate their network requirements and desired network behavior to the SDN Controller via a northbound interface (NBI).
- Router – a networking device that forwards data packets between computer networks. Routers perform the traffic directing functions on the Internet.
- Load Balancer – a device that acts as a reverse proxy and distributes network or application traffic across a number of servers. Load balancers are used to increase capacity (concurrent users) and reliability of applications.
- Switch – a multiport network bridge that uses MAC addresses to forward data at the data link layer (layer 2) of the OSI model. Some switches can also forward data at the network layer (layer 3) by additionally incorporating routing functionality. Such switches are commonly known as layer-3 switches or multilayer switches.

Motivation for Implementation

Consider the five main component types:

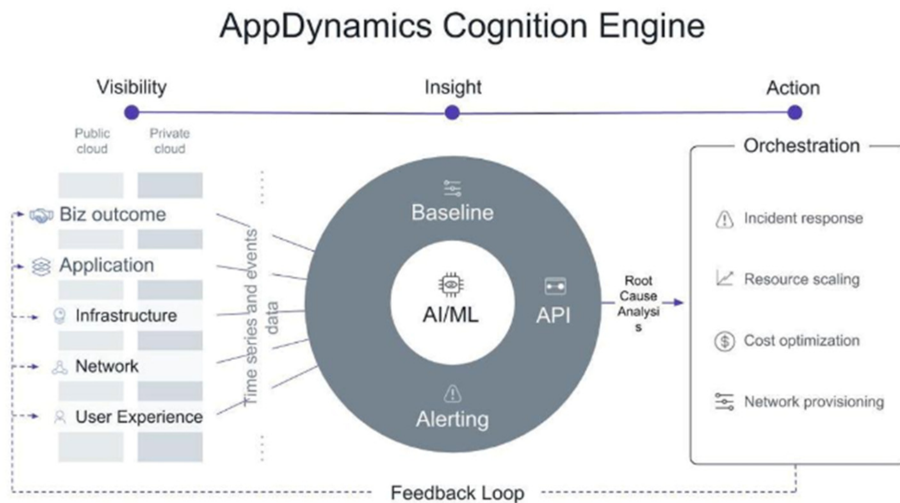
- Application: application level - code level instrumentation;
- Endpoint: OS level - mainly kernel level instrumentation;

- Network: Wire level - mainly layers 1-7 of the protocol stack;
- Container: specific to container/mainly peer to peer communication; and
- Browser: user level - mainly javascript instrumentation.

The Application and Network components are very good at collecting metrics that indicate how well things are performing. Further, the APM sits inside the application and is able to expose critical information such as the user involved, whether errors occurred, etc. that can be shared with the network infrastructure and NPM tools so that the network team has visibility into the application stack to correlate it to what is seen on the network. In addition, many of the metrics/metadata can be used to make important QOS decisions by NIMs that have the ability to optimize traffic. The ability to share information across all of these component types in real time would be extremely valuable and make correlation much easier than just shipping four different sets of metrics to a backend to figure out how they will be correlated.

<u>Central Nervous System</u>

A Central Nervous System (CNS) is a concept of a system that brings these components together. The "visibility" label in the CNS shown below closely describes the five instrumentation viewpoints.



The described messaging system can be used as part of a CNS to provide the technology to tie together the Visibility components to provide the Insights and Actions. This provides a path for direct correlation of transactions for the Visibility components

versus having to correlate in a centralized backend. More specifically, the described messaging system has the ability to "share" the "Insight" between the components before being sent to the "Insight" backend and in fact can act as a "peer" to "peer" shared system to create a richer set of metrics or alerts, etc. than simply pushing metrics into an "Insight" engine. The "Action" function can also be achieved in a "peer" to "peer" manner based on a CNS Policy stored in the members. The messaging system easily plugs into the CNS model and actually extends it such that the feedback loop is "distributed" versus solely "centralized."

Operation of Communication

Unlike normal communications that are client/server based, the communication described herein is tailored around "indirect" transmissions seen by all enabled components versus something like a standard REST request and response. It is more of a "distributed" communication that is focused on "networked inline" members and/or multicast, similar to broadcast. This form of communication reduces the amount of transmissions and eliminates the need to know IP addresses or even where the information is being sent.

Essentially, the communications are designed around either/or the following transmission mechanisms:

- Injecting Headers: insertion of the messaging as HTTP Headers into either the request or response of the transaction; and
- Multicast Beacon: multicast UDP Transmission of the messaging as a "beacon" which can be seen by anything.

For Injecting Headers, the advantages over direct communications are:

- eliminating the need for a separate transmission;
- automatic transaction correlation (the injected response *is* the correlated transaction); and
- eliminating the need for transmissions to all other members, since this is a broadcast for any member "in line" to the transaction path.

For Multicast Beacon, the advantages over direct communications are:

- eliminating the need for transmissions to all other members, since this is a broadcast;

- automatically sent to all members on the network; and
- is UDP so no connection necessary.

The messaging is built around a Key Value Pair (KVP) concept. Basically, the KVPs correlate to the message sections and follow the following format:

Every message contains the following:

name_DeviceName = name

name_MessageType = message

Each message contains a set of key/values depending on the Message Type. The Unique Name prefixes all keys, which allows for multiple members to add their own messages to a transaction response.

However, there are advantages and disadvantages as to which of these methods to use:

- Insertion of HTTP Headers: great if not encrypted and if the other ACICP(s) one wants to see the messaging is "in band (inline)," as it piggybacks an existing transaction making it easier to correlate.
- Multicast UDP Transmission: great to be seen by "out of band (not inline)" and is clear text, but is a separate transmission.

Message Types

Basically, although the messaging is very flexible and extensible, currently there are five message types involved:

1. Service Announcement

A service announcement is a multicast UDP message sent every *x* seconds announcing that a service is available that understands the ACICP. All information is basically built around Key Value Pair (key = value).

- DeviceName = name
- MessageType = Announce
- ACICP Component
    - Type = APM|NPM|LB, RTR|SDWAN|SDN|Container
    - Vendor = vendor
    - Version = version

- OS Runtime Info
  - OS = os
  - Version = version
  - Memory = memory
  - CPUs = cpus
- Application Runtime Information (Optional)
  - Platform = JVM|.NET|PHP|Python|Node.js| Go
  - Version = version
  - Vendor = vendor
  - Memory = memory
  - Type = Tomcat|Websphere|ASP|etc.
  - Addresses = address, address, etc.
- Container (Optional)
  - Type = Docker|Kubernetes|ServiceMesh
  - Version = version
  - Vendor = vendor
  - ManagementIP = mgmtIP
  - InternalAddresses = address, address, etc.
  - ExternalAddresses = address, address, etc.
- Custom (Optional)
  - name = value

2. Event Message Type

The event message type may be sent out in real time and may be grouped/batched by transaction.

- DeviceName = name
- MessageType = Event

Status Event

- Type = StatusChange
- SubType = Online|Offline

Error Alert

- Type = Error
- Component = Cache|Wire|Database|etc.
- Description = blah...blah...blah
- Impact = blah...blah...blah
- ErrorCode = code
- User or IP = user or ip
- Code Trace = stack trace

Performance Alert

- Type = Performance
- Component = TransactionLatency|NetworkResets|etc
- Description = blah...blah...blah
- Impact = blah...blah...blah
- MetricName = metric
- MetricId = metricId
- MetricValue = value
- MetricThreshold = threshold
- User or IP = user or ip
- Code Trace = stack trace

3. Collaboration (Request/Response - sent in real-time)

- DeviceName = name

Request (for transaction context information must be on inbound transaction)

- MessageType = Request
- FromDeviceName = name
- Request = user|role|topology|metric|etc
- Id = requestid

Response (to the Request)

- MessageType = Response
- FromDeviceName = name
- Response = response

- Id = requested

4. Metrics (broadcast every *x* minutes)

DeviceName=name

MessageType=Metrics

Recurring (for each metric) - Metrics are "Deltas" over the reporting interval

MetricName_1 = name

MetricMin_1 = min

MetricMax_1 = max

MetricAvg_1 = avg

MetricStdDev_1 = stddev

.... recurring ....

MetricName_n = name

MetricMin_n = min

MetricMax_n = max

MetricAvg_n = avg

MetricStdDev_n = stddev

5. Application "Inline" Response for HTTP (injected in real Time)

DeviceName = name

MessageType = AppResponse

Latency = latency

ResponseType = BT|Proxy|etc

Errors = error, error, etc.

Priority = 1-10

Custom (Optional - example is for AppDynamics)

BTName = name

Application = appName

Tier = tier

Node = node

Id = "singularityHeader" (a correlation header)

Differentiation from Existing Messaging Technologies

The idea of putting information into an HTTP Header or Parameter is known, but mostly as a proprietary method for a vendor, generally passing a very limited number of parameters, and to a single known device from the same vendor which would process that data. All other devices would simply ignore the information.

The described technology is different from other device-to-device messaging in the following ways:

- it is full duplex (send and receive);
- it can co-exist with encryption: using the UDP multicast messages can be "broadcast" in clear text;
- it is one to many (multicast) messaging ("beacon");
- it connects all vendors and device types (it is intended to be a standard);
- it can be used to set QOS downstream by any devices capable;
- it does not require the need to know a destination; and
- it is "connectionless."