

JINSPIRED Developer

Software Metering, Metric Monitoring and Event Signaling for Java Developers

[Builds](#) [Usage](#) [Docs](#) [Repo](#) [API](#) [Install](#) [Config](#) [Console](#) [Site](#)

Introduction

There have been numerous attempts to introduce some manner of resource management, in particular reservation based, into enterprise Java applications for the purpose of (*supervisory*) control and quality of service (QoS). Unfortunately most have not met with much success in part because they typically required a complete rewrite of both runtime and application libraries. In addition such proposed solutions failed to effectively address how resource consuming activities could be distinguished from the current thread of execution which was seen as the primary consumer. Likewise both service classification (*class of usage*) and fine grain resource usage profiling were absent along with how such proposed resource management frameworks could be integrated into a much wider QoS solution that was application (*context, activity*) and cost (*meter*) aware.

A primary reason for such approaches not achieving traction has been that they effectively ignored the many years of experience in doing resource management in the networking field which today is the main adopter and innovator in this space.

This article puts forward the argument for applications to be viewed more like the underlying networking layer, adopting many traffic engineering and quality of service optimizations techniques. It presents a mapping guide from one domain to another and then shows how this has been implemented in a solution that offers a production QoS solution for applications developed in Java, Scala, JRuby/Ruby and Jython/Python as well as all other JVM languages and eventually other runtimes/platforms including Microsoft's .NET/Azure.

QoS for Networks

Lets first start with a standard definition of QoS:

"The ability of the network to provide better or "special" service to a set of packets/dataflows to the detriment of other packets/dataflows".

More formally Quality of Service (QoS) is a set of service level requirements to be met by a network in transporting a dataflow. This contractual commitment is based on quality metrics such as bandwidth (*throughput*), delay, delay variation (*jitter*), packet loss and availability which is indirectly managed by controls (*classification, scheduling, policing/shaping and queuing*) put in place to meet the other metrics.

From a resource perspective QoS is the division of network resources in a non-equal manner providing more to some and less to others.

Traffic Characteristics

QoS in a networking context looks at four traffic characteristics:

1. **Bandwidth:** The number of bits per second that can be expected to be transmitted.
2. **Delay:** The elapse time between when a packet is first sent and when it arrives at its destination.
3. **Jitter:** The variation in the arrival rate, or delay introduced, of packets sent at a uniform rate.
4. **Packet loss:** All routers lose, drop or discard packets for a number of reasons including congestion, corruption, rejection, fading, errors and faults.

Traffic Planning

There are 3 basic steps in implementing QoS at the network layer:

1. Identify traffic and its requirements in terms of service levels
2. Divide and associate traffic into classes of service levels
3. Define QoS policies for each class of service level which cause some change in the traffic characteristic mentioned previously.

Traffic QoS Service Classification

QoS service classifications involves differentiating one packet from another by examining fields inside of the packet's header. In some cases this classification is performed automatically and adaptively based on traffic patterns and resource consumption behavior.

QoS service classification is important in providing a level of indirection and mapping to service level policies and the traffic control mechanisms used by such policies.

Traffic QoS Congestion Management

Congestion is typically managed using queues and schedulers which split traffic across queues and schedulers based on service classification associated with a packet or flow. Queuing and scheduling generally occur when there is congestion and/or resource capacity needs to be protected, reserved and managed.

Queue management and queue scheduling builds on the traditional first-in-first-out (*FIFO*) queuing, which provides no service differentiation, by enqueueing and scheduling across different queues with different buffer sizes, queue lengths and transmission rates.

Traffic QoS Policing and Shaping

Policing at traffic management control points is based on traffic contracts which define how much data can be sent. When measurements made by the policing component exceed a defined threshold packets are dropped.

Shaping is similar to policing but instead of dropping packets it tries to conform the data rate to the traffic contract for the service class. Shaping works by smoothing the peaks and troughs of data transmission in order to optimize or guarantee performance and bandwidth.

Both policing and shaping measure the rate at which data are sent. When an overflow occurs, the traffic rate does not conform, in such cases the policer will drop a packet and the shaper will delay the sending of the packet. Policing discards and shaping delays sending. Both are responsible in ensuring the data rate don't exceed a specified threshold defined in a contract.

QoS for Applications

Lets now rewrite our standard definition of QoS for networking in the context of application runtimes:

"The ability of the runtime to provide better or "special" service to a set of calls/threads to the detriment of other calls/threads".

Call Characteristics

QoS in the context of an application runtime looks at the following request/call characteristics:

1. Throughput: The number of requests/calls per second that can expect to receive a valid and completed response.
2. Response time: The time between a request/call is sent (*or received*) and when it's response is received (*or sent*).
3. Response time variation: The variation in the response time of requests/calls. In non-deterministic runtimes with delay components such as thread scheduling, garbage collection, and concurrency controls there will always be variation.
4. Exceptions & Errors: Faults can occur in runtimes such as timeouts caused by contention and heavy workload concurrency on resources both local and remote.

Call QoS Planning

The QoS planning steps for an application runtime can now be stated as:

1. Identify request/call patterns and their requirements in terms of service levels.
2. Divide and associate request/call patterns into classes of service levels.
3. Define QoS policies for each class of service level which cause some change in the request/call characteristic mentioned previously.

Call QoS Service Classification

For QoS service classification differentiation is typically done by examining the execution context associated with a request/call. This is generally done at the request/call entry points – top of "request" call stack or method.

How a thread is distinguished from another is by some aspect of the execution context associated with a thread. Examples of such context include:

- ThreadGroup and Thread
- HTTP Request and/or User/Session
- Executing Code and/or Caller Chain

QoS service classification in application runtimes associates a service class with a thread. This association can be performed multiple times removing previous classifications and adding additional service classifications as the execution proceeds and unlike in typical networking environments a thread can be associated with one or more services at a particular execution points in its processing.

Call QoS Contention Management

Contention (*congestion*) is managed using concurrency and control constructs such as semaphores/latches/locks which are generally backed by implicit or explicit queues consisting of parked threads during contention. Queuing and scheduling will only come into play when there is contention and/or when resource capacity needs to be protected, reserved and managed. Backing queues can be fair or non-fair both of which can support bargaining. At each managed method call or execution point the current or most applicable service classification will be used in targeting (*selecting*) which semaphores will need to be accessed and permits acquired before proceeding in the call processing. For reservation based queues a determination of the number of permits needed also needs to be made.

Call QoS Policing and Shaping

Policing is based on request/call contracts which define how much requests/calls can be sent. When measurements made by the policing component exceed a defined threshold requests/calls are rejected by way of an exception or error code.

Shaping is similar to policing but instead of rejecting requests/calls it tries to conform the request/call rate to the request/call contract using some form of delay mechanism such as waiting and parking.

Both policing and shaping measure the rate at which requests/calls are called. When an overflow occurs, the request/call rate does not conform, the policer will reject (*throw an exception or return an error code*) a request/call and the shaper will delay/suspend/park the processing of the request/call. Both are responsible in ensuring the request/call rate don't exceed a specified threshold defined in a contract. Both are commonly referred to as request/call throttling or request/call rate limiting.

In implementing both policing and shaping a leaky bucket algorithm is applied to a gauge based on an analogy of a bucket that has a hole in the bottom through which any water it contains will leak away at a constant rate, until or unless it is empty. Water can be added intermittently, i.e. in bursts, but if too much is added at once, or it is added at too high an average rate, the water will exceed the capacity of the bucket, which will overflow.

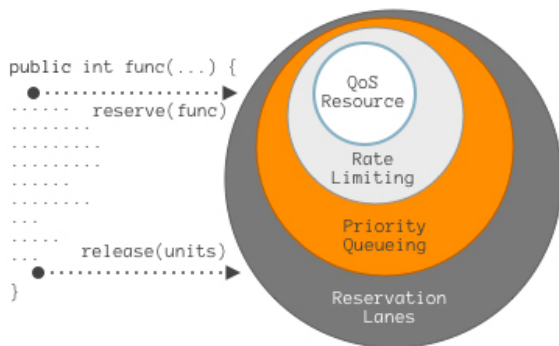
Below is a table which provides a useful but somewhat loose mapping of networking components and concepts to application runtime components and concepts.

Network	Runtime
Packet	Request/Call
Flow	Thread
Router	Method
Datalink	Call Site
Buffer	Resource/Meter
Queue	Semaphore

QoS for Applications Implementation

The notion of an application being a network is the basis for JXInsight/OpenCore's "QoS for Apps" technology. QoS is layered on top of OpenCore's software activity based metering and costing runtime whereby all metered code execution points (*methods, dynamic-language call sites*) within an application's codebase are modeled as virtual network traffic management devices that can perform service (re-)classification, (re-)prioritization, resource reservation, traffic/throughput rate limiting along each of the (*predicted*) code execution paths and across threads and reservation lanes.

JXInsight/OpenCore allows QoS to be injected transparently into an application without any code changes. First calls are placed into the entry and exit points of a method or call site by an instrumentation agent. The calls inserted begin and end named activities which are represented are referred to as probes in OpenCore. When the underlying implementation of a probe is created by the metering engine additional capabilities beyond the basic metering support can be added by installed probes provider which could be viewed somewhat like decorators. QoS is one such capability that can be installed with a few system properties without any changes to the instrumentation performed by the agent.



JXInsight/OpenCore's QoS implementation is based primarily based on resource reservation and pools. QoS "resources" are defined using the following system property pattern:

```
jxinsight.server.probes.qos.resources=${resource},${resource},...
```

For each resource defined a capacity can be defined.

```
jxinsight.server.probes.qos.resource.${resource}.capacity=...
```

QoS resources can be viewed as global named semaphores with the capacity being the number of permits.

QoS services are defined using the following system property pattern:

```
jxinsight.server.probes.qos.services=${service},${service},...
```

QoS "services" are then associated with probes (i.e. methods) or partial probe name (packages, classes).

```
jxinsight.server.probes.qos.service.${service}.name.groups=...
```

By default services will be associated with each global resource though it is possible to define which resources are applicable and as well as creating new local resources specific to the service class.

```
jxinsight.server.probes.qos.service.${service}.resources=...
```

QoS "queues" can be defined to front resources during reservation.

```
jxinsight.server.probes.qos.resource.${resource}.queue.fifo.enabled=true
jxinsight.server.probes.qos.resource.${resource}.queue.priority.enabled=true
```

Multiple queues can even be defined using "lanes" which can place additional capacity limits on resources for a particular service class or promote or demote the prioritization of the reservation request needed when priority queuing is enabled.

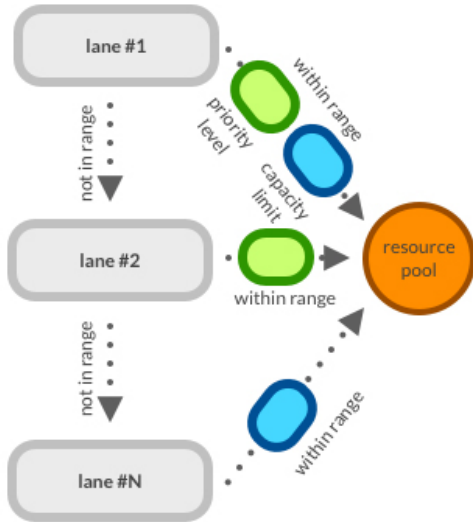
```
jxinsight.server.probes.qos.resource.${resource}.lanes=${lane},${lane},...
jxinsight.server.probes.qos.resource.${resource}.lane.${lane}.capacity=
jxinsight.server.probes.qos.resource.${resource}.lane.${lane}.priority.level.promotion=0..7
jxinsight.server.probes.qos.resource.${resource}.lane.${lane}.priority.level.demotion=0..7
```

When a reservation is made by a service it is mapped to a lane based on the "lower" and "upper" reservation thresholds defined for each lane.

```
jxinsight.server.probes.qos.resource.${resource}.lane.${lane}.lower=
jxinsight.server.probes.qos.resource.${resource}.lane.${lane}.upper=
```

QoS shaping referred to as rate limiting in JXInsight/OpenCore, is also supported for both resources and reservation lanes with the specification of a "limit" and "interval".

```
jxinsight.server.probes.qos.resource.${resource}.rate.limit=
jxinsight.server.probes.qos.resource.${resource}.rate.interval=
jxinsight.server.probes.qos.resource.${resource}.lane.${lane}.rate.limit=
jxinsight.server.probes.qos.resource.${resource}.lane.${lane}.rate.interval=
```



The elements of the QoS model defined up to now come into play when a method is entered (or a probe begun) just like in the networking layer when a packet arrives. First the metering runtime checks whether a QoS service has been associated with the method (or probe). If a service is mapped then for each of the QoS resources associated with the service a reservation is made. The reservation can pass through shapers, lanes, and queues before finally arriving at the resource. Once all reservations have completed the execution of the method proceeds. When the method does complete the capacity reserved during the entry phase is returned to each resource.

How a service determines how much capacity is requested from each resource during the reservation phase is defined by a "reservation" property on the resources themselves which is then adjusted by the amount of capacity already reserved by callers that have also made a reservation against the resource. Possible values include "one", "meter", "inc", and "lease".

```
jxinsight.server.probes.qos.resource.${resource}.reservation=
```

With "one" only a single unit of capacity (or permit) is reserved from the resource's pool. The "inc" reservation method is similar to "one" in that only a single unit is reserved but different in that it does not adjust for units already reserved by its callers.

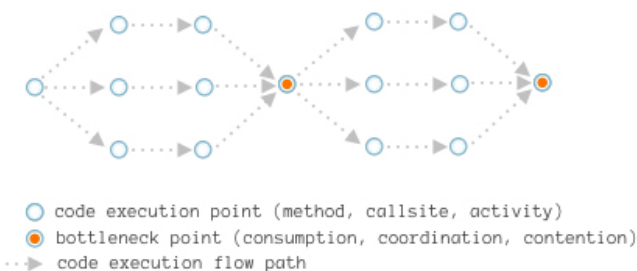
Both "meter" and "lease" require a meter to be associated with a resource which can be specified as follows:

```
jxinsight.server.probes.qos.resource.${resource}.meter=
```

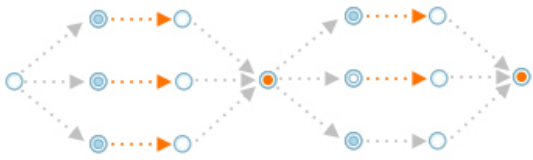
With "meter" the resource uses the metering profile of the method (or probe) to determine the units of capacity. If a resource is a clock based meter then methods which are slower in executing will request more capacity from the resource's pool. With "lease" the resource uses the metering usage up to the point of the request starting from some outer caller point as the units of capacity to be reserved.

The Application is the Network

Today's application runtimes operate very similar to networks where packets are calls and routers are methods and callsites.



In engineering resilience and self-regulation into application runtimes, largely driven by today's devops movement, developers should look to apply tried and tested quality of service techniques to their request processing pipelines, paths and flows.



- ⦿ QoS service (re-)classification point
- ⦿ QoS reservation extension or (re-)prioritization point
- QoS queuing, prioritization, rate limiting

Note: The content here was originally published on [InfoQ](#) under the article title [QoS for Applications](#).

Search