

# Leveraging a Messaging Infrastructure for Global Load Distribution

Co-authored by  
Marco Malva (Tibco)  
Brian Husted (Pyxis)

## Background & Objectives

### *Executive Summary*

Distributed computing is an emerging technology trend that attempts to provide a commodity based and infinitely scalable computing environment. As bandwidth and networks continue to grow exponentially, enterprise organizations are demanding new capabilities to maximize the computing capacity for large scale applications. Multiple data centers, a messaging backbone, and complex event processing provide the foundation to implement these capabilities within a wide-area network. The messaging backbone provides the asynchronous transport to connect geographically dispersed data centers and the underlying business services. Complex event processing provides an intelligent approach to monitor, track and provision the distribution of processing. This paper focuses on industry best practices for employing a global messaging solution to federate geographically dispersed processing centers hosting large numbers of resource intensive services.

### *Use Case Overview*

The use case discussed in this document contains several data centers connected by a network of Tibco Enterprise Message Service (EMS) brokers. Data centers are geographically dispersed and differ in processing capacity, capability and variety of services offered. The resulting heterogeneous distribution of resource intensive services in a high-volume environment imposes a significant architectural challenge for the routing and load balancing of service requests.

Each data-center consists of 4 layers:

- a) The **storage** layer providing physical transparent access to the payload
- b) The **routed messaging** layer for connecting the data centers
- c) The **non-routed** messaging layer for connecting distributed service instances with asynchronous messages placed in queues
- d) The **processing** layer hosting services.

This document assumes that the EMS infrastructure is deployed in a highly available and fault-tolerant configuration as described in the EMS user documentation.

## *Technical Summary*

One of the simplest approaches to achieve load balancing in a wide-area network (WAN) would leverage proxy queues (aka global queues) to distribute processing to multiple backend data centers. This approach would allow all of the local JMS queues (aka home queues) within each data center to appear as one global queue advertising all the messages to any subscriber within the WAN. This enables the system to distribute load based on the underlying processing capacity at each location. The benefits of this approach include minimized complexity, self throttling, consumer driven load balancing, and fault tolerance all of which are automatically provided by the messaging infrastructure. However, in general, it is preferable to publish but not consume high volumes of messages from proxy queues as the acknowledgment must be processed by the home queue. It is also possible to consume from a home queue while the route is disconnected, but this is not the case for proxy queues.

To address the inefficiencies of consuming from proxy queues, an alternate approach is required to minimize network communication while preserving a consumer driven approach to load distribution. JMS message timeouts provide a basic capability to determine when a processing center is over tasked or is unable to process requests. Expired messages would be published onto a global “load shedding” queue that is visible to all data centers with idle capacity. Leveraging message timeouts as an indicator of degradation in service level agreements provides a simple approach that can minimize the volumes of messages advertised to the WAN. Complex event processing would provide a more sophisticated load shedding algorithm that is capable of correlating numerous metrics such as queue depths and CPU utilization. CEP could also be leveraged to share state among data centers to determine the existence of idle capacity within the WAN before shedding requests. In other words, if there is no idle processing capacity within the WAN then the requests should remain local to prevent recursive shedding. The approach discussed is intended to optimize the network bandwidth and EMS performance by reserving the more resource intensive proxy queues to handle the intelligent routing of messages to idle processing capacity.

The actual consumption of messages for processing is handled in a separate messaging layer that is local to each data center. To be effective, expired messages must contain a selector or the originating jms destination name to bridge the message from the “load shedding” proxy queue to the local destination where the services are consuming. This consumer driven approach allows each data center to pull additional work rather than pushing messages around the WAN which can lead to inefficiencies in processing latency, network utilization, and EMS performance.

The following sections provide the technical detail related to implementing a distributed messaging solution.

## **Technical Topics for EMS**

- ❖ EMS monitoring
  - ❖ EMS load sharing or load shedding
- a) As long as a request is processed within an acceptable time-frame, i.e. the goal is not to equally use all available processing capacity but to ensure that all available resources are utilized to maximize throughput while optimizing resources.
  - b) The EMS infrastructure provides the asynchronous transport to connect services while the data payload, which is much larger, is handled by a distributed storage layer.
  - c) While the storage layer abstracts from the physical location of the stored data there is a geographical implied cost to access more remote data, i.e. it is optimal
  - d) In other words as long as local resources are sufficient to serve all requests it would be sub-optimal to shift requests to other data-centers.
  - e) The term 'load shedding' is used to describe the desired system behavior more accurately.

### *Functional modules To Implement load shedding*

- detecting that load-shedding is required, i.e. processing times are approaching unacceptable values
- identifying where to direct load to, i.e. intelligent load routing based on service availability and resource utilization
- physically routing requests from one data-center to another

## **Undesirable System State**

- load shedding in an overloaded global system will further degrade available processing capacity
- Avoid the “bad” message, i.e. be prepared to handle messages that never get processed. This both increases latency and network bandwidth usage by continuously republishing messages. This could occur if a service is requested but does not exist at any of the data centers. These messages would continue to timeout and ultimately flood the network. \
- Avoid orphans, i.e. avoid requests that get directed from one site but the link or the site experiences a downtime and the request gets backlogged and site experiences a failure and hence requests are not processed

Each of the above points is discussed in greater detail in the following sections.

## Identifying Processing Latency & EMS Monitoring in General

There are a couple of ways to detect processing latency, ranging from the most simple form of EMS infrastructure monitoring up to tracing requests end-to-end.

### *EMS Server Monitoring & Configuration*

It is generally advisable to monitor the following key metrics on an EMS server:

- **number and/or of pending messages** -- commonly used to generate an alert when pending messages reaches a threshold this is more sort of a catch-22
- **memory utilization** – commonly used to generate an alert when memory utilization reach in combination with server setting: `max_msg_memory` and `route_recover_interval` and `route_recover_count` for EMS connected via routes it is also advised to set the `reserve_memory` server parameter if virtual memory is limited and malloc failures possible – should be at least of size of largest message if that is practical the maximum memory for statistics should also be limited (`max_stat_memory`)
- **number of connections** – commonly used to watch out for rogue clients in combination with server setting `max_connections` to avoid running out of file-descriptor which is much worse as it impacts administrative connections, too inbound and outbound message rate – commonly used to identify unusual high or low message rates or situations where outbound >> inbound the outbound >> inbound is a situation typically encountered where an application (e.g. TIBCO BusinessWorks) relies on `session.recovery()` for error handling but the error is data related. The identification of ‘unusual’ high or low message rates requires either business derived thresholds or historic data – both often more challenging to obtain and manage.
- **CPU utilization** – a continuous CPU utilization of about one core often indicates some sort of problem
- **Size and utilization of the data-storage files** – often combined with manually scheduled compact operation on the data-storage files
- **Flow control** – This should be enabled on the server as it throttles fast producers even if the destination is not flow-controlled. However, enabling flow-control on a destination is often not a recommended unless the sending application does not mind its thread being blocked on the send activity which is the case for some batch-oriented applications.
- **Multiple listen ports** – Most interesting if one wants to prepare for a possible EMS server separation or there are reasons to bind to multiple NICs.
- **EMS buffer tuning** – This will be addressed in a separate discussion thread as it is a topic in its own and was discussed over lunch.

## EMS Destination Monitoring & Configuration

It is generally advisable to monitor the following key metrics on JMS destinations (queues/topics):

- **Number and/or of pending messages** -- commonly used to generate an alert when pending messages reaches a threshold this is a more specific monitoring than on the server level. In reality the challenge is to come up with good values as a too low value results in too frequent (false) alerts and a too high value will result in too late alerts. If used to guard against resource misuse then it is sufficient but if intended to monitor for prompt message processing it is a rather poor approach.
- **Number of consumers/durables on a destination** – commonly used to generate an alert if less than a minimum number of consumers are connected or to identify when a new durable consumer was created. This sort monitoring can ensure that a minimum processing capacity is ‘online’ which does not necessarily result in timely processing though. However the opposite is true, i.e. with non sufficient processing capacity – or as the extreme cases zero – the requests are unlikely to be processed in a timely fashion.
- **Inbound and outbound message rates** – this can be used to predict backlog growth or shrink and how long messages will be queued based on current performance
- **Maxmsgs and maxbytes** -- This can be specified on a destination and the overflowPolicy (discardOld, rejectIncomming) specify what to do in case that the limit is reached. The specific semantics depend on the destination type and the message delivery mode and the EMS documentation should be consulted for more details. This is commonly used to limit a certain queue or topic to a maximum size in order to limit side effect on other destinations. Please note that even when not set at some point the EMS server will reject new messages as either the message memory will be insufficient or the disk storage. However if such EMS wide limit is reached all destinations are impacted.
- **FlowControl** – blocks the sending thread if a consumer is online until the limit is no longer reached. A good setting for performance benchmark as it is an easy way to throttle any kind of producer but for production systems only certain classes of producers deal well with blocked threads.
- **Expiration** – override any message level setting and results in a zero-expiration for any message delivered to the consumer, i.e. the server honors it but if it is already handed over to the consumer then it will no longer expire. When a message expires it is either destroyed or, if the JMS\_TIBCO\_PRESERVE\_UNDELIVERED property on the message is set to

true, the message is placed on the undelivered queue so it can be handled by a special consumer. See Undelivered Message Queue for details. The message expiration was discussed as a possible option to shed events that are not processed in a timely manner.

- **Prefetch** – the ems server delivers to each consumer  $1/2N$  to  $N$  messages ( $N$ =prefetch value). This is to optimize the message delivery and reduce latency by avoiding clients waiting for a message being requested and send by the EMS server to the consumer. The default work fine for most message sizes though smaller messages benefit from larger values and large messages may require lower values if memory is a concern.
- **ConsumerInfo.Details class** – allows to get information about the milliseconds elapsed since the last sent and acknowledge on a consumer level, i.e. it requires online consumers. In that case these values can be used to approximate the time a message spends between being send and acknowledged. Note that sending a message to a consumer is subject to the pre-fetch setting which means that if right now  $N$  messages are pre-fetched no more messages will be sent unless  $N/2$  message have been acknowledged. In any case it is a good metric to detect a blocked consumer or a consumer-side dropped message.

### ***EMS System Monitor Messages for Message Process Tracking***

It is possible to leverage system monitoring messages to trace when a message is received by the EMS server, sent by the EMS server to a receiver and acknowledged by that receiver or when a message is expired or otherwise discarded by the messaging system. These system messages are presented to the `$sys.monitor.{q|Q|t|T}.{r|s|a|p}.{destination name}` and discussed in more detail in appendix B of the EMS user guide.

The lower letter system monitor (q,t) do receive only the message header including the message ID and message correlation ID while the upper letter (Q,T) do receive a full blown copy of the message. For the purpose of tracking message processing the lower-letter are sufficient if the message correlation ID is populated with a unique value by the source system and copied along by all processing components. It is obviously less expensive and less of a security exposure to just copy the message header than the full body.

If an EMS would be operated close to its capacity of handling inbound or outbound network packets system monitor that track every message with an additional 1 to 3 messages will prevent the EMS from handling the volume. If an EMS operates well below its network processing capacity the overhead of generating these system monitor messages can well be taken without impacting

the message delivery of the payload messages. For all intermediate systems it is sufficient to trace the receive event as the next processing step will also result in a receive event. For the last system the acknowledgement should also be traced to verify the timely consumption by the last system. Thus the overall outbound message volume of the EMS server is doubled if not tripled.

The correlation of the tracked messaging steps is something for what TIBCO Business Events is well suited, as well as for the application of SLA and the reaction if SLAs are missed or likely to be missed. In fact for load shedding not the individual SLS miss is of interests but the time-wise accumulation of them as that would indicate a processing bottleneck requiring load shedding.

### ***Periodic Echo Event***

The idea is to send an 'empty' echo event through the system to measure its processing time. This requires co-operating message consumer that do not choke on the echo event but process it correctly allowing measuring the wait-time.

### ***Summary***

The most accurate monitoring of processing times can be achieved with EMS system monitors but this is also the most complex solution in terms of development effort. Less accurate but with identical reaction are the approaches with a periodic echo event and queue depth monitoring. Each of the approaches has as a counter reaction the instantiation of a consumer that sheds a certain percentage of the pending message of the impacted queue.

An alternative approach that was to leverage the message expiration to automatically redirect message that waited for an undue time for their processing to the dead letter queue. From there the message would be picked up by the shedding process that based on the current system load in the various data-centers would re-publish the message to either the local or any other data center.

**The appeal of this solution is the reduced complexity.**

## **Load Shedding – Load Balancing & Load Redirection**

### ***Load Shedding***

A key aspect of load shedding is to identify how each data center will receive backlogged requests from other data centers. In a consumer driven approach, each data center would make the decision to accept additional tasking or continue to only accept requests from the home queue. A global topic could be used to distribute the health and status of each individual data center. Complex event processing would provide situational awareness within each data center by correlating data from the global topic. Each data center can leverage rules and ontology to react to determine the idle processing capacity. Leveraging many of the metrics discussed within this paper including EMS queue depths provide the

metrics to optimize processing distribution. By externalizing thresholds and conditional values used within the rules, system operators can adapt to runtime conditions to more intelligently route processing.

Using this approach, all the data-center monitoring must do is publish regularly (e.g. every minute), a list of queue/topic, their current backlog (and size) and number of online consumer. This is the information collected as part of standard EMS monitoring and simply needs to be packed into a JMS topic message.

### **Calibration**

Another key point is the calibration of the redirection of messages from one data center to another. It is important to record metrics on the number of requests that are redirected among data centers. This will be necessary to measure the effectiveness of the load balancing strategies and to prevent over propagation of messages and moving the data. JMS message timeouts can be used to indicate an SLA violation which triggers expired messages to be published to a global queue that is visible to data centers with idle processing resources.

An important item is to monitor how messages are being delivered or processed, i.e. can they expire and thus result in duplicate processing. Duplicate processing is generally not desirable but an overload situation can drive a situation from bad to worse. In addition the shedder (dead letter consumer) should be throttled to avoid overrunning the other data centers. This can be either static (maximum rate) or dynamic (maximum backlog / flow control / route flow control).

In an SLA monitoring based solution a new, dynamic consumer must be instantiated once an overload situation is detected. That consumer must be paced to avoid re-direction to few or too many requests. Again this can be done static (minimum and maximum rate) or dynamic (original queue backlog and shed target queue backlog). Once the overload condition is no longer present the shedding can be discontinued and the consumer be destroyed.

### **Physical Routing**

The last key point is the physical routing of shedding request. The discussed alternatives were:

- a) global queue with home in the data-center that has the overload situation
- b) global queue with proxies in the data-center that has the overload situation
- c) non-global queue with remote connections

In general it is preferable to publish **and not to consume from proxy queues** as the acknowledgment must be processed by the home queue anyway. It is possible to consume from a home queue while the route is disconnected, that is not the case for proxy queues.

In solution (a) the appropriate proxies and routes must be configured to enable the flow between data centers that can help each other. The nice thing is that a flow control on the home queue would automatically throttle the dead letter reader to the pace of all attached consumers, i.e. a self regulating system. Of course that would require a suitable thread model in the dead letter reader such to avoid side effects between different interfaces, i.e. at least one publishing thread per home queue. Ideally one consumer with selector per home queue with its own session such to leave messages of too slow home queues on the dead letter queue.

The 'other' data-center can then bring their consumers online as they wish to help. This means that in each data center for each interface there are:

- a) The normal request queue used for its own original requests
- b) A global queue where to put data to when some of the own request must be shed
- c) A proxy queue for any other data-center that offers the same service such to be able to help said other data-center.

Assuming 5 data-centers that are  $2+5=7$  queues per every service and  $1+5=6$  consumers, and every additional data center changes the layout and requires at least a configuration change in the application to now also serve the new proxy queue. This could become a maintenance problem. This could be mitigated by a local dispatcher that reads all the proxy queues of the other data-center (of which hopefully only one or two has any data anyway because once more than  $\frac{1}{2}$  of the data-centers start shedding there may be a wide spread overload situation which may best be left alone). The downside of this is that now the proxy queues could be read at a high speed which means the reading must be throttled to avoid sucking in all the requests. Again flow control might be suitable but that requires a shed target queue, i.e. a queue where this dispatcher places the proxy requests into. That way the normal request queue operates without flow limit as that would impact the local requests, too. But the requests obtained from other data centers are not flow-controlled plus additional not dispatched at all if there is no consumer on the shed target queue (requires additional logic).

The solution (b) is pretty much like (a) but avoids consumption from a proxy queue [which is generally recommended] but also requires the dead letter reader to decide which data-center to direct request X to. Once the request is re-placed it is stuck there, i.e. the solution is slightly less dynamic and changes in the throughput after the re-placement are not taken into consideration.

This means that in each data center for each interface there are:

- a) the normal request queue used for its own original requests
- b) a global queue that is used as target by other data centers when these other data-center have something to shed
- c) a proxy queue for any other data-center that offers the same service as a place to put the own requests that are shed

This design already features only 2 queues from which to read (a) and (b), i.e. one queue for the own original requests and one queue for shed requests by any other data center. Adding more data-centers does not change the consuming application.

A max messages limit could be used on (b) to start backlogging at the shedding data-center(s) once a certain request pile has been 'sucked in'. That would have to be combined with proxy queue depth monitoring such to throttle the dead letter queue reader avoiding to place all timed out requests pre-maturely.

The biggest downside of this solution is that if a route to a specific data-center is offline \_after\_ requests were place into the proxy queue but not yet transmitted. Those would be stuck until the route is online again. It is open to tests (support ticket) if messages stuck in a proxy queue can be expired or otherwise retrieved while the route is offline.

The solution (c) is like (b) but replacing proxy queues with direct connections by a dispatcher component. This solution does not have the downside of message stuck in a proxy queue but requires a dispatcher component that like the emsd would connect to all the other data-centers.

### **Priority & Other Flags**

It was suggested to increase the priority of expired requests to give them precedence. This only applies if the same queue is used for normal and expired requests which also assimilates with the not the case of any of the above designs which is mostly a consequence of an easy-to-implement flow control. However there are ways to implement flow control also when the original and expired request are placed in the same queue which is when the priorities should change. In order to allow priority within normal requests a range of priorities should be defined early on as there are only 10 priorities [0 to 9].

An expired request should probably carry a flag indicating that it was shed and an information field about the source (data-center) of the original request as well as its timestamp. Those could prove useful in monitoring and trouble shooting.

### **Data Center Downtime**

The data-center downtime can be mitigated by deploying the dead letter queue reader into each of the legs within the data center (storage layer, message layer, processing layer) and thus provide fault-tolerance by redundancy. That works nicely with all approaches, e.g. once the original request expires they will be re-directed to the other data-centers.

In fact once could also deploy an on-demand shedder that does not read from the dead letter queue but from the original request queue and such avoid to have

wait for the normal timeout. This is an appropriate intervention in a planned downtime of the processing layer where one does not want to intervene in the storage layer but simply wants to redirect all requests for the duration of the maintenance.