Benchmarking WebSphere MQ may at first seem a daunting task; however, with some basic process in place it can be easily managed. In this whitepaper we will divide the benchmarking process into three parts: Establishing the definition of the benchmark, running tests and comparing results and deriving conclusions.

Defining the Benchmark
When you begin to define the benchmark, start by specifying explicit performance targets and key performance indicators (KPI’s). Recommended KPI’s include expectations for: message rate, message size and target response and round trip time. Plan on using multiple topologies for your benchmark. Start with a simple topology with a single queue manager and then utilize a more complex one with a several queue managers and channels. Utilize both persistent and non-persistent messaging. Next, define the process and tools that will be used to run the benchmark. Finally, measure the results, compare to prior runs and adjust accordingly.

First Benchmark Test
This is only an example for illustrative purposes. It is important to recognize that results will be different as your benchmark will be specific to your requirements such as the number of messages, message size, target systems and other variables.

For the first benchmark example, we will establish a baseline and attempt a performance target of over 1,000 messages per second. We will use 100 byte messages with a single server running one physical CPU with 4 cores. The server is configured with 8 GB memory and is utilizing SSD drives. We will start a single queue manager running on a single local queue.

We suggest you utilize a tool to generate messages. We recommend the free download on nastel.com, MQSonar® as the tool to generate messages. In order to measure how quickly applications are de-queuing messages off the queue, in this test we run with “confirm on delivery” and “confirm on arrival” report messages as well as measuring message latency on the queue itself.

The concept is to measure the average round trip for a message and the message latency. The average round trip time is measured by taking the average of the sum of each of each of the individual steps. The average message latency is measured by calculating the difference between the time the message arrives on the queue, (you receive a confirmation on arrival report) and the time the message is de-queued and delivered to the application (confirm on delivery).
While a single queue manager benchmark may seem trivial, knowing the basic performance signature is important for understanding the impact of changes you make over time, as compared to this baseline.

\[
\text{Avg. Round trip} = \text{AVG}(\text{SUM}(T(1) + T(2) + T(3) + T(4)))
\]

\[
\text{Avg. Msg. Latency} = \text{AVG}(\text{SUM}(T(2) - T(1)))
\]

**Figure 1:** Benchmarking a single queue manager

**Process**

As shown in Figure 1, perform the following steps.

Step 1: Use MQSonar to put a message on the target application queue.

Begin by starting a command prompt on a box where MQSonar is installed.

Next, create a local command queue and use MQSonar to send the first of 1,000 messages to the application queue.
Step 2: The application receives the message.

Utilize a test application to receive the message by reading from the local application queue.

Step 3: The application responds back immediately with a response onto the reply queue.

In our case, we used the MQSonar ping application to send a response back to the reply queue.

Step 4: The response message will be received by MQSonar.

MQSonar is reading from the local reply queue.

Results

![MQSonar test results](image)

**Figure 2:** MQSonar test results – 1st run
In Figure 2 we see the results from this test which took just over 21 seconds to complete. We achieved a rate of 235 messages per second round trip, but if we take the report messages out, it is actually about 141 messages a second. Out of total time (21 seconds), 14 seconds were spent doing PUTS and about 7 seconds were spent on GETS. The average PUT rate was about 70 messages per second and the average GET rate was at 576 messages a second. Due to an imbalance of PUTS vs. GETS, you should run the same exact test a few more times and see if you can achieve consistent results.

In Figure 3, our second run, we achieved better results. The total time was approximately 15 seconds.
The effective round trip was about 200 messages a second averaging 2.8 seconds per message. There was still a difference between the PUT and GET rates. It took about 12 seconds to do the PUTs and a total of 3 seconds to do the GETs.

The propagation time was 2.9 seconds and the reflection time was 1.7 seconds.

**Propagation** time is how long it takes for the message to actually arrive to the destination to be picked up by the application.

**Reflection** time is how long it takes the application to respond back to the message.

These are excellent indicators to understand if the time is being sent on the outbound or inbound side. These become more meaningful when dealing with multiple queue manager configurations as discussed below.

An additional key item to look at is Message Latency. Message Latency is only computed when you have configured “confirm on arrival” and a “confirm on delivery”. In this test, these specific flags are enabled. Similar to delivered and read receipts in email, these provide insight into how long after the message was received it was read off the queue. The difference between when the message arrives on the queue and when it is actually delivered can indicate contention or dispatches issues for the responding application. In this test, the average latency was just over a second. But it is an average and since we put 1000 messages, some messages were probably processed quickly while others sat on the queue longer. This is also demonstrated by the arrival rate in this test, which was 78 messages per second and the delivery rate which was 64 messages per second.

**Multiple Queue Manager Benchmark Tests**

Another type of benchmark will be using the same set up as the previous benchmark using MQSonar as a tool and running 1,000 messages with a size of 120 bytes. However in this case, instead of using a single queue manager, we use multiple queue managers and channels. Using 2 queue managers connected via channel, this benchmark will, as a result have more steps.
Figure 4: Topology of multi-queue manager benchmark

**Process**
As shown in Figure 4, perform the following steps.

**Step 1:**
Use MQSonar to put a message on the remote queue, which actually places it on a transmission queue.

**Step 2:**
The sending channel reads the message of the transmission queue sends to the remote queue manager. The receiving channel puts the message on the local queue.

**Step 3:**

\[
\text{Avg. Round trip} = \text{AVG} \left( \text{SUM}(T(1) + T(2) + T(3) + T(4) + T(5) + T(6)) \right)
\]

\[
\text{Avg. Msg. Latency} = \text{AVG} \left( \text{SUM}(T(3) - T(2)) \right)
\]
The application receives the message.

Step 4:

The application responds back immediately with a response onto the reply queue.

Step 5

Just as in the inbound message, the response travels over the channel, and into the reply queue.

Step 6

The response is consumed by MQSonar.

When running the multiple queue manager tests, use a burst of 1,000 messages and confirm on delivery and confirm on arrival to see where the time is spent. Be sure that your channels are up and running. If not, expect some delay in propagation or reflection times for initial runs.

![Command Prompt](image)

**Figure 5**: Results for the second benchmark using multiple queues
It is probably safe to say that your expectation is that the more complex topology should be slower than a single queue manager test.

![Command Prompt](image-url)

**Figure 5**: Results for the second benchmark using multiple queues

**Results**

The total PUT time is 6 seconds and the total GET time is 13 seconds. The round trip was approximately 20 seconds. So contrary to our expectations above, the run time was actually less than the single queue manager. What was expected was that the multiple queue manager benchmark would run slower than the single queue manager benchmark because it involved channels and has a more complex topology. There are contributing factors that help explain this, the fact that the messages were small and sent in bursts, that additional processing threads were involved and so on. However, the purpose of using the benchmark is not always to explain the results, but rather to provide a point to compare the impact of changes to the environment, intentionally or unintentionally, over time.
In fact, comparing benchmarks to one another while interesting is not the primary goal, the benchmarks compare a given result for a given configuration at a given point in time.

**The Real World**
These numbers are best case since when you are running benchmarks you are testing the “plumbing” and not processing the business logic. Your applications will most likely achieve lower throughput than these tests due to business logic constraints and contention on queues.

When setting up the benchmark, model them as closely as possible to your own applications. For example, increase message sizes to be representative of your messages, decrease the number of messages, or set the persistence value.

You can also run multiple configurations at the same time to emulate different application workloads.

**The impact of Change**
Change can be expected over time, either due to changes you initiate or changes required by the applications or WebSphere MQ itself.

![Channel Configuration, shown in Nastel AutoPilot for WebSphere MQ](image)
For example, one change you might consider is adjusting the batch size for the channels or changing the non-persistent Message speed for the channel (which sends non persistent messages outside of sync point). These could impact the throughput of the channels. Making changes and comparing test results will provide insight into your environment and the impact of these changes.

Another example of a change that can have impact on your applications is the use of persistent or non-persistent messages. While both fall under the commit control, non-persistent messages are not logged and will perform faster.
As you can make changes to the environment, review the results and how they compare to the baseline.

Maintaining history is an important part of this process. While the command tools do not maintain history directly, the information captured during the run can be exported to a file or database. Another option to collect history is to integrate MQSonar with Nastel's AutoPilot for WebSphere MQ which can store and monitor the data being analyzed.
Nastel AutoPilot for Middleware

Nastel Technologies provides middleware-centric application performance monitoring solutions for mission-critical applications and is the only monitoring vendor with a unified platform to support all software and appliance-based middleware technology requirements: **WebSphere MQ, TIBCO EMS** and **RV, Solace, WebSphere DataPower, WebSphere Message Broker** and home-grown.

**AutoPilot® for WebSphere MQ** handles all the aspects of WMQ management including:

- **Self-Service** - Provide your stakeholders with self-service access to WebSphere MQ with full control over what they can see and do via safe and secure role assignment and policy. Self-service access uses a standard web browser without a requirement for a local installation.

- **Agents or Agent-less** – Choose which configuration is best for your needs - both configurations are available

- **Monitoring** — Deep, native monitoring using agents or agent-free providing auto-discovery of all WMQ objects and their state. Collects management data, statistics and events. WMQ message tracking (transactions)

- **Administration** — command and control of all WMQ resources, e.g. start/stop/create/delete queues, channels, messages, DLQ rerouting, message management, object replication and authorization control

- **Configuration Management** — maintain all WMQ objects and their definitions in a SQL database, commands for change, compare, rollback, auto-generation of alter events when changes are detected

- **Security** — Consolidate all access via a single web interface, Kerberos authentication and integration with Active Directory

- **Corrective Action** — policy-driven, initiation when WMQ problems are identified
Scale - Nastel AutoPilot is monitoring many of the world’s largest WebSphere MQ environments and can scale almost linearly over its active data grid.

Message Tracking – automatically track messages and transactions end-to-end, both synchronous and asynchronous. Message stitching can be based on message payload.

Visit Nastel Technologies at www.nastel.com for more information.