WebSphere MQ for z/OS V7.0
Performance Report

Version 1.0

August 2008

Tony Sharkey

Take Note!

Before using this report please read the general information under "Notices"

First Edition, August 2008

This edition applies to Version 7.0 of WebSphere MQ for z/OS.

Notices

This report is intended for Architects, Systems Programmers, Analysts and Programmers wanting to understand the performance characteristics of **WebSphere MQ for z/OS V7.0**. The information is not intended as the specification of any programming interfaces that are provided by WebSphere MQ. Full descriptions of the WebSphere MQ facilities are available in the product publications. It is assumed that the reader is familiar with the concepts and operation of WebSphere MQ.

References in this report to IBM products or programs do not imply that IBM intends to make these available in all countries in which IBM operates.

Information contained in this report has not been submitted to any formal IBM test and is distributed “as is”. The use of this information and the implementation of any of the techniques is the responsibility of the customer. Much depends on the ability of the customer to evaluate these data and project the results to their operational environment.

The performance data contained in this report was measured in a controlled environment and results obtained in other environments may vary significantly.

Trademarks and service marks
The following terms, used in this publication, are trademarks or registered trademarks of the IBM Corporation in the United States or other countries or both:

Enterprise Storage Server
IBM
SupportPac
WebSphere
WebSphere MQ
z/OS
zSeries

Other company, product and service names may be trademarks or service marks of others.

Summary of Amendments

Date	Changes
4 August 2008	Initial version

Table of Contents

Performance Highlights – WebSphere MQ for z/OS v7.0

This report focuses on performance changes since the previous version (V6.0) and on the performance of new function in this release.

SupportPac MP16 "Capacity Planning and Tuning For WebSphere MQ for z/OS" will be updated to include WMQ Version 7.0 information. MP16 will continue to be the repository for ongoing advice and guidance learnt as systems increase in power and experience is gained.

Existing Function

General Statement of Regression

CPU costs and throughput are not significantly different in version 7.0 for typical messaging workloads.

In a request/reply model, the new scavenger process has proven beneficial with smaller messages – for example, putting and getting 2KB non-persistent messages saw an 2.5% increase in throughput and an 8% decrease in the cost of processing a message.

Client queuing, where a client application is putting messages to a queue or getting messages from a queue sees an improvement in the processing cost per message.

Client messaging, where a client application is putting messages to a queue, a server application is getting the message and putting a reply to a common reply queue, which is then retrieved by the client using the correlation-id also sees a noticeable reduction in the cost per message.

Above Bar Usage

Since z/OS V1.2, 64-bit virtual storage has been available within a single address space. This 64-bit storage is also known as "above the bar" storage. The "bar" refers to the 2 GB line that was the 31-bit virtual addressing limit. The introduction of z/OS V1.5 allowed for shared 64-bit virtual storage.

WebSphere MQ V7.0 is the first MQ release on z/OS to begin to exploit this feature.

In version 7.0, two areas of the queue manager have been changed to exploit the 64-bit "above the bar" storage.

- Intra-Group Queuing (IGQ) buffer.
- Pub/sub related function.

In WebSphere MQ Version 6.0, APAR PK45456 ensures that the 100MB buffer allocated for IGQ is only allocated provided the queue manager is part of a Queue Sharing Group.

Version 7.0 goes a step further, in that it only allocates the 100MB IGQ buffer if the queue manager is part of a QSG and then ensures that the storage is allocated above the bar, allowing the storage below the bar to be used for buffer pools, message handles etc.

Above the bar usage in the Pub/Sub related function will be discussed in the section titled "Above the Bar usage in Pub/Sub"

ECSA Usage

The extended common storage area ("ECSA") usage between WebSphere MQ for z/OS version 6.0 and version 7.0 has increased for an idle queue manager by 1.5MB.

When running with 9,000 clients connected, the delta for the ECSA in the queue manager has also increased by 70 bytes for each client.

Memory Allocation

In version 6, each 4KB page in each bufferpool was separately allocated, whereas in version 7, the queue manager allocates each bufferpool in a single request.

This has a number of effects:

1 Queue manager restart time is reduced
2 The queue managers' real storage footprint is reduced, when long term storage protection is set on by the WLM "storage critical" option.

Channel Initiator Enhancements

The channel initiator now produces a message reporting the state of memory usage, similar to the queue managers CSQY220I message.

This new message "CSQX004I" takes the form of:

```
+CSQX004I @qmgr CSQXSPRM Channel initiator is using 29 MB
of local storage, 1444 MB are free
```

The message is logged at channel initiator start and then either every hour if the usage does not change or when the memory usage changes (up or down) by more than 2%.

A second change to the channel initiator involves the use of buffer pools for each dispatcher task. These are not the same as the queue manager's buffer pools. The channel initiator buffer pool functionality allow a varying size of buffer to be allocated depending on the size of the message that is being put or gotten and results in a buffer much closer to the true size of the message.

Should the channel initiator use 80% of its available memory, it will attempt to release unused buffer pools.

New Function

Above the Bar usage in Pub/Sub

The new publish/subscribe code is written using 64-bit addressability. This means there is virtually no hard limit on the number of topics and subscriptions that can be defined to a queue manager.

Not all object information used for publish/subscribe-type processing can be retained using only above the bar storage. For example, when a local queue is defined to a z/OS queue manager, a record is written to PAGESET(0). Similarly, when an administrative topic object is defined, a record is written to PAGESET(0) and there is also information stored above the bar. Upon queue manager restart, this above the bar data needs to be rebuilt from the persisted PAGESET(0) data.

Administratively defined Topics

Administratively defined topics only need to be defined when a topic tree or part of a topic tree needs to have specific non-default attributes. For example, in order to keep part of the topic tree closed to specific users, the topic node may be set to reject wildcard subscriptions. As such, it is not expected to have significant numbers of topic nodes defined administratively (e.g. via MQSC commands). The section titled "Administratively defined Topics" shows how defining topics can affect queue manager restart times.

How much storage does an administratively-defined Topic use?
A Topic object has the potential to be costly in terms of pageset usage, primarily because the topic string attribute "TOPICSTR" may be a maximum of 10KB in length.

Topic String length	PAGESET(0) usage (KB)	Below the Bar usage [1] (KB)
Short (50 bytes)	2	6.6
Med (1800 bytes)	4	14
Long (9500 bytes)	12	36

This sizing means that for 10,000 topic objects that have been administratively defined with short topic strings,

- PAGESET(0) usage will require 20MB (or 5000 pages).
- On top of this, there will be additional memory costs below the bar of 66MB
- In addition, 53MB above the bar storage will be used.

Since the "above the bar" storage is not restricted in a practical sense, the 10,000 topics with short topic strings will cost the queue manager 86MB of its available extended private storage.

[1] Excluding pageset usage.

Message Properties

When using the MQSETMP verb to add message properties to a message, the message data will grow accordingly.

How much storage does a message property use?
When a message is published with properties, the message on the queue will contain the message properties even if the subscriber has requested PSPROP(NONE) and the target queue has PROPCTL(NONE).

Only when the message is actually retrieved from the queue will the message properties be formatted into the subscribers required format.

Do message properties affect the size of the message being got?
When a message is put with properties attached, the message is held on the queue in an internal format such that the message is larger than just the message data plus the MQ headers. This also means that the setting of the PSPROP attribute on a subscription does not affect the size of the message held on the queue.

When attempting to get a message with properties, there are several different cases to be aware of:

1. Getter application uses MQCRTMH to create a message handle which can then be used to set, inquire or delete the properties. The MQGET will return a message the same size as the original putter application message buffer.
2. Subscription has PSPROP(NONE) and target queue has PROPCTL(NONE). In this case there will be no properties associated with the message on the get, so the getter application will receive a message buffer of the same size as the putter applications' message.
3. Getter application gets the message from the queue, where the properties are still embedded in the message buffer. This will require additional parsing to get to the actual message data. In this case, different PSPROP values on the subscription will change the size of the message buffer returned to the getting application.

To explain scenario 3 further, a topic "RAINBOW" has been defined with 5 subscribers – the first four having different PSPROP options – NONE, MSGPROP, RFH2 or COMPAT. The fifth subscription has PSPROP(NONE) and the destination queue also has PROPCTL(NONE), e.g.

```
DEF TOPIC(RAINBOW) TOPICSTR('/RAINBOW1') DURSUB(YES) REPLACE

DEF SUB(PS1) TOPICOBJ(RAINBOW) DEST(LQ1) PSPROP(NONE)
DEF SUB(PS2) TOPICOBJ(RAINBOW) DEST(LQ2) PSPROP(MSGPROP)
DEF SUB(PS3) TOPICOBJ(RAINBOW) DEST(LQ3) PSPROP(RFH2)
DEF SUB(PS4) TOPICOBJ(RAINBOW) DEST(LQ4) PSPROP(COMPAT)
DEF SUB(PS5) TOPICOBJ(RAINBOW) DEST(LQ5) PSPROP(NONE)

DEF QL(LQ1) DEFSOPT(SHARED) SHARE REPLACE STGCLASS(REMOTE)
DEF QL(LQ2) DEFSOPT(SHARED) SHARE REPLACE STGCLASS(REMOTE)
DEF QL(LQ3) DEFSOPT(SHARED) SHARE REPLACE STGCLASS(REMOTE)
DEF QL(LQ4) DEFSOPT(SHARED) SHARE REPLACE STGCLASS(REMOTE)
DEF QL(LQ5) DEFSOPT(SHARED) SHARE REPLACE STGCLASS(REMOTE) +
  PROPCTL(NONE)
```

```
DEF QA(RAINBOW) TARGET(RAINBOW1) TARGTYPE(TOPIC)
```

If a 100 byte message were to be published to the topic “RAINBOW” with a property name of “COLOUR” and a property value of “BLUE”, the 5 messages that gotten from the queues would be returned to the application with different buffer lengths. The following table shows the expected message data sizes.

PSPROP value on subscription	Message Buffer – required length (includes 100 bytes for actual message)
NONE	172
MSGPROP	240
RFH2	384
COMPAT	294
NONE Queue is defined with PROPCTL(NONE)	100

How many properties can be added to a message?
WebSphere MQ Version 7.0 has been tested to support in excess of 32,000 properties on a single message, but from a practical point of view, it is expected that a message will contain no more than 10 properties in a single folder.
The chart in section “How many properties can be added to a message?” shows the increasing cost as more properties are added to a message using the MQSETMP verb.

Subscriptions

How big is a subscription?
A subscription that has been created via an application program can specify all of the following MQSD attributes with lengths of up to 10KB:- SubName, SubUserData, SelectionString and ResObjectString. This means there is potential for subscriptions to be large compared to other MQ objects. From an administration perspective, a subscription name of 10KB is not going to be ideal to manage, so reaching the upper limit length-wise should be rare.

A subscription with a SubName length of 30 bytes will use 830 bytes of pageset storage once the subscription has been consolidated. Each of these subscriptions additionally used 4KB of “above the bar” storage.

For the most simple subscription case, the consolidated subscription length is 770 bytes plus the length of the ObjectString (or topic string) being subscribed to plus the length of the subscription name.

Durable subscriptions are stored on messages on a queue named “SYSTEM.DURABLE.SUBSCRIBER.QUEUE”.

How long will my queue manager take to restart?
It is envisaged that there will more subscriptions to topics than topic admin nodes by an order of magnitude. As such, it is paramount that a queue manager restart time is not significantly affected by large numbers of subscriptions. One way this restart time

has been reduced is by consolidating durable subscriptions so that there is less overhead when reading the queue containing durable subscriptions upon queue manager restart.

Non-durable subscriptions that exist on queue manager shutdown will not be present upon queue manager restart. As such, all measurements relating to subscriptions in this section are using durable subscriptions only.

The subsection "How long will my queue manager take to restart?" in section "Subscriptions" show the CPU cost of queue manager restart with increasing numbers of durable subscriptions.

When does consolidation occur?
To reduce the queue manager restart time when there are significant numbers of durable subscribers, a consolidation process has been introduced. This process can be seen to be running in the queue managers' log by the following messages:

```
CSQM075I @qmgr CSQMDURR Consolidation of durable subscribers
started
CSQM076I @qmgr CSQMDURR Consolidation of durable subscribers
finished
```

The consolidation process is triggered when there are 100 durable subscriptions are added to the "SYSTEM.DURABLE.SUBSCRIBER.QUEUE". The process then attempts to consolidate these subscriptions into as few 50KB messages as possible.

The running of the consolidation process does not preclude further subscriptions taking place.

What affects the maximum number of subscriptions for a queue manager?
When durable subscriptions are made, a message is written to the subscription queue ("SYSTEM.DURABLE.SUBSCRIBER.QUEUE"). As previously mentioned, when 100 subscriptions have been added, these subscriptions are consolidated into fewer, but longer, messages on the same queue.

By default, the subscription queue is defined to storage class "SYSLNGLV" which in turn is defined to PAGESET(2). As more durable subscriptions are defined, the pageset usage increases. An example of this usage can be seen in the section "Pageset usage with Durable Subscriptions". In addition, the subscription information is held in "above the bar" storage. An example of this usage can be seen in the section entitled "Above the Bar usage with Durable Subscriptions".

Publish

Is publish more expensive than put?
Compare a put to queue with the simplest case of publish to a topic with a single durable subscriber, where the subscriber is not in an MQGET-with-wait.

This is a simple scenario to set up and can be used with a legacy application to exploit the new publish/subscribe capability e.g.

```
* Define a local queue
DEF QL(TARGETQ) DEFSOPT(SHARED) SHARE PROPCTL(NONE)
* Define a topic
DEF TOPIC(ATOPIC) TOPICSTR('/ATOPIC') DURSUB(YES)
* Define a subscription to the topic
DEF SUB(TO_ATOPIC) TOPICSTR('/ATOPIC') DEST(TARGETQ) +
  PSPROP(NONE)
* Define an alias to the topic that points to the target queue
DEF QA(TOPICALIAS) TARGET(TARGETQ)
```

This configuration allows the application to either put directly to the local queue "TARGETQ" or to put to the alias queue "TOPICALIAS", which will use the pub/sub engine to route the message to "TARGETQ".

The path length for the pub/sub route is longer and as such will be more expensive but as more subscribers are added, the cost for the publish decreases.

In the current implementation MQ on z/OS does not cache the handles to the subscribers. This means that for each put to "TOPICALIAS", the pub/sub engine has to look up all subscribers to the topic and for each subscriber has to open, put to and close the queue.

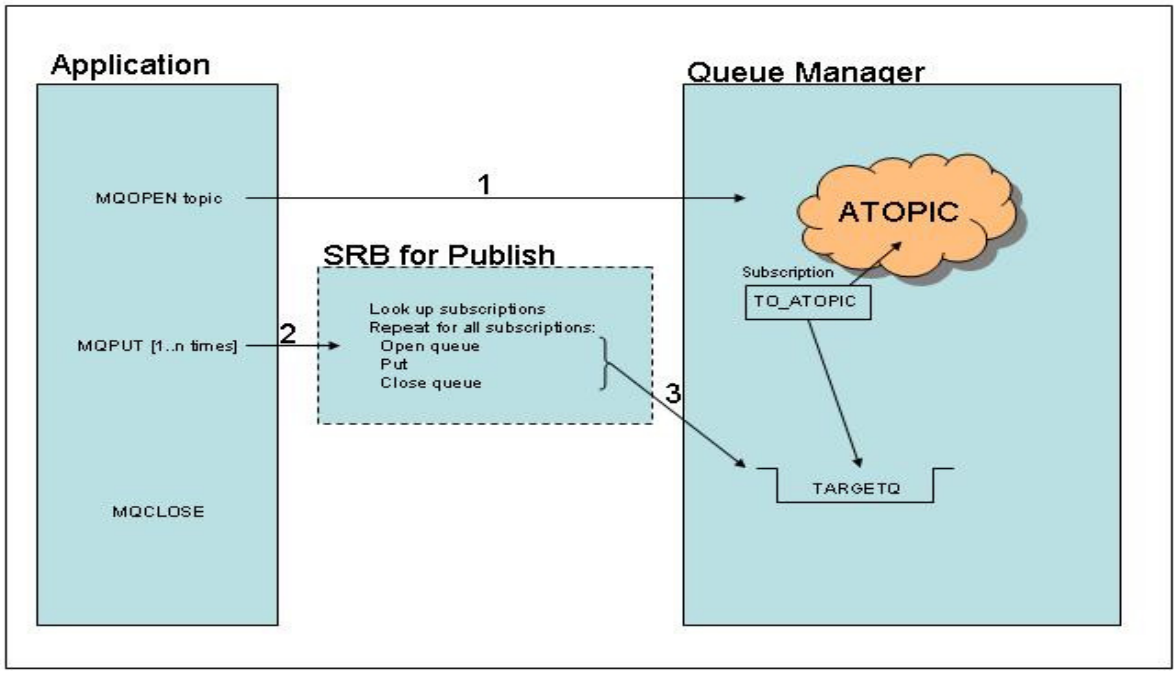
Do message properties affect the cost of publish to topic?
Adding properties to a message using the MQSETMP verb increases the size of the message put to a queue. As the message size increases, the size of data written to the queue also increases, which in turn means a longer time to write the message to the queue.

For up to 32 properties, the cost of publishing a message with properties is not noticeably more expensive than publishing a message without any properties. As the number of properties increases, the CPU cost for SRB used for publishing increases.

Accounting for the cost of a Publish
When a message is published to a topic, either directly or indirectly via an "alias-to-topic" queue, the queue manager needs to ensure that the message is delivered to all the subscribers or none of the subscribers. To do this, MQ schedules a preemptible SRB[2].

MQ schedules the SRB to run under the publishers address space. This means that for a topic with many subscribers, the MQPUT can take a long time to respond as the SRB will attempt to put a message to all of the target queues before returning.

[2] An SRB is a "Service Request Block" that represents a dispatchable unit used to perform a particular function in a specified address space.



The cost of the work performed by this SRB is charged back to the client address space and accounted depending on the publishing environment:

1. **Batch** - The publisher application is charged with the costs incurred by the SRB and as such the CPU cost will be included in the step and job totals as reported by the IEF374I message, rather than the queue managers' address space.
2. **CICS** – The CICS address space will be charged for the work the SRB performs rather than the individual transaction. CPU costs incurred by this SRB will be recorded in the MQ SMF (type 116) records.
3. **Client** – The channel initiator will incur the cost of the SRB when a client application is publishing to a topic.

Publish/Subscribe

In a Publish/Subscribe environment there are two components – publish and subscribe. The performance of publish to an administratively defined subscription has been described earlier in this document. This section will introduce subscription via an application and publishing to that subscription and the cost of getting the published message.

How much does a Subscribe cost from an application program?
When using an application to make a subscription, a decision needs to be made as to whether the subscription is durable or non-durable and whether to let the queue manager manage the destination of published messages.

All of these options have benefits so it is important to make that decision for each individual case.

Some questions that may influence a decision are:-

- Do the subscriptions need to be retained over a queue manager restart?
- Is the subscriber application going to disconnect and re-connect at a later point and still require the messages that were published whilst it was not connected?
- If using a client to subscribe, is the network unreliable, prone to outages etc?
- Is every published message important to the subscriber?
- Does the administrator want to be concerned with managing the destination queues?

The performance section of this document gives a breakdown of the associated costs for subscribing via an application.

When issuing the MQSUB verb, MQ schedules an SRB to ensure that the subscription and all associated topic tree changes will be completed or backed-out as one unit of work.

Specifying Message Properties at Time of Subscription

WebSphere MQ version 7.0 on z/OS supports the use of message properties when subscribing to a topic. This allows the application to rely on the queue manager to filter out unwanted messages and only be sent relevant messages.

Messages properties are specified at the time the MQSUB is issued using the MQSD "SelectionString" attributes. The syntax of a message selector is based on a subset of the SQL92 conditional expression.

Some simple examples of selection strings are:

- COLOUR='RED'
- (COLOUR='BLUE' OR COLOUR='RED')

There is a small overhead associated with the cost of specifying message properties at subscription time but it is not expected that an application will be making many subscriptions.

Publishing when subscribers specify message properties

When a subscriber specifies message properties, the publisher of any message will incur an additional cost when it puts the message. This cost will be incurred by the SRB that runs in the publishers address space.

The cost incurred varies depending on the selection criteria. For example, if a message is published with 128 properties and one subscriber is selecting on the first property and another subscriber is selecting on the last property, the publisher will see a higher cost associated with the second subscriber. Details of the cost variances can be seen in the performance section of this document.

Similarly, for more complex selection criteria, such as "PROPERTY1='MQ' AND PROPERTY2='CICS', the cost to the publisher increases.

Wildcards

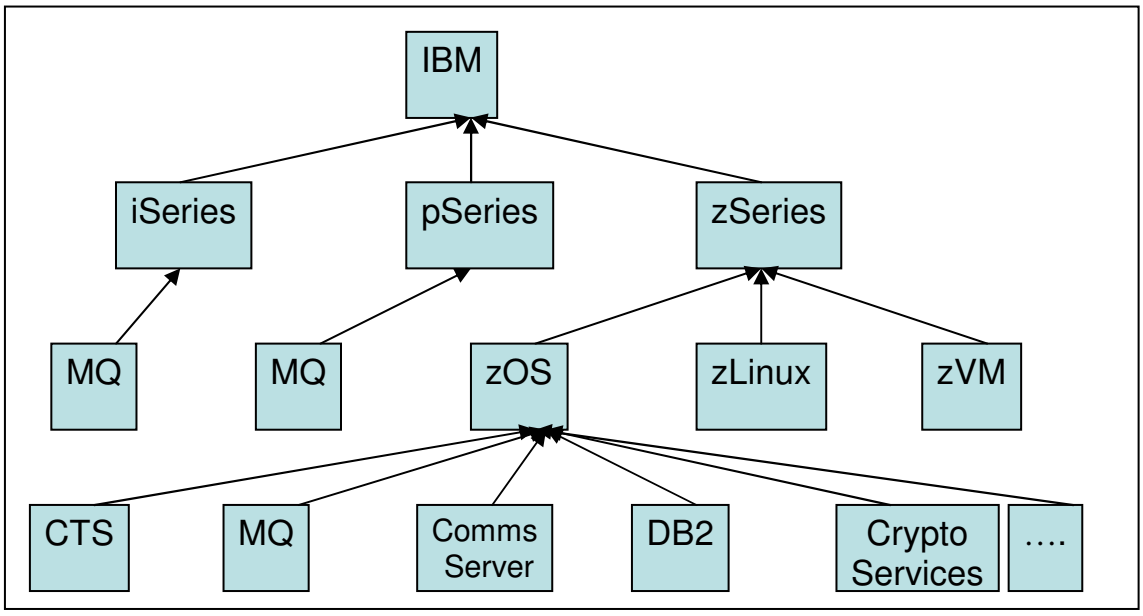
The use of the wildcard characters as part of a subscription allows an application to subscribe to multiple topics in a single subscription, rather than issuing multiple MQSUB verbs. As there is no explicit subscription name, there is the potential to subscribe to more topics than required and as a result, some unwanted messages may be received.
The cost of subscribing with a wildcard is negligible as is the cost of publishing when the subscribers have specified wildcards in their selection criteria.

There is a balance that needs to be determined, potentially on an application by application basis -

- Is the application interested in all messages for a topic or just some of them?
- Can the topic be wildcarded to the required level of granularity and still be useful?
- Will the application always be interested in all the messages that match the wildcarded topic name?
- Is it difficult to filter out unwanted messages in the application?

From a programming perspective, it may be simpler to request all of the messages for a particular set of topics rather than issuing many MQSUB verbs, but there may be information overload (i.e. too many messages) and the publisher incurs a cost for each subscriber, which means the unwanted messages are paid for twice – once by the publisher and then again for the subscriber who has to determine that it doesn't need this message.

Taking a topic tree such as the one below:



If an application is interested in all "MQ" messages it can subscribe using wildcards and will receive all messages published to "IBM/iSeries/MQ", "IBM/pSeries/MQ" and "IBM/zSeries/zOS/MQ".
Similarly if the application is interested in all subjects relating to "IBM/zSeries/zOS" then wildcards are again easy to use and may be the best solution.

However, if an application is only interested in messages published to “IBM/zSeries/zOS/CTS”, “IBM/zSeries/zOS/MQ” and “IBM/zSeries/zOS/DB2”, the cost to subscribe explicitly to the 3 topics is far less than the cost of doing a wildcard subscribe to “IBM/zSeries/zOS/+” and filtering out the unwanted messages such as those published to “Comms Server”.

Server-Connection Attribute SHARECNV()

The SHARECNV attribute is only available on a server-connection channel and specifies the maximum number of conversations that can be sharing each TCP/IP channel instance.

A value of 0 provides no sharing of conversations over a TCP/IP channel instance. This means the channel does not use the new WebSphere MQ Version 7.0 function with regards to:

- Read Ahead (queue attribute DEFREADA)
- Client Asynchronous Consume
- Heartbeating

A value of 0 does not preclude the use of the asynchronous put request, as specified by the queue (and topic) option DEFPRESP(ASYNC).

A value of 1 allows no sharing of conversations over a TCP/IP channel instance but does allow channel heartbeats whether in an MQGET call or not. It also supports read ahead and client asynchronous consume.

A value in excess of 1 allows sharing of conversation over a TCP/IP channel instance.

High SHARECNV limits have the advantage of reducing queue manager thread usage when the client application is threaded. However, if a large number of conversations sharing a socket are all busy, there is a possibility of delays as the conversations contend with one another to use the receiving thread.

Asynchronous Put

In WebSphere MQ Version 7.0, you can choose to put a message to a queue or topic using the MQPUT or MQPUT1 verb without the application having to wait for the queue manager to complete the call.
This can be achieved by:

- The application setting the MQPMO_ASYNC_RESPONSE in the PMO
- The application specifying MQPMO_RESPONSE_AS_Q_DEF when putting to a queue/topic that has been configured to have DEFPRESP(ASYNC).

By using asynchronous puts, the MQPUT/MQPUT1 verb is not waiting for a response from the queue manager. This means that the client application is not dependent on the queue manager sending a confirmation-of-receipt message. On a slow or heavily utilised network, the client application will find these MQPUT/MQPUT1 verbs completing much faster than before.

If an MQ generated Message-ID is required, the MQ Client will require a response from the queue manager once in every 256 messages.

Read-Ahead DEFREADA(YES | NO)

Read ahead can be used on a client to allow non-persistent messages to be sent to a client without the client application having to request the messages.

Typically, when a client requires a message from the server, it sends a request to the server. For each message required, a separate request is made.

Using read ahead can improve performance when consuming non-persistent messages from either MQI or JMS applications. When read ahead is enabled, messages are sent to an in-memory buffer on the client, called the read ahead buffer.

The client will have a read ahead buffer for each queue it has open with read ahead enabled. The messages in the read ahead buffer are not persisted. The client periodically updates the server with information on the amount of data in the read ahead buffer.

Read ahead is not available when the SVRCONN channel attribute SHARECNV is set to zero.

Asynchronous Consume

"Asynchronous Consume" is a way to allow an application to register an interest in messages and identify a call-back routine which will get invoked when a message arrives. This gives the following benefits:

- The application can continue processing non-MQ related work without the message arriving, instead of the thread remaining in an MQGET-with-wait call.
- The call-back routine invoked as part of the message arriving will be given a message buffer of the correct size for the message.
- The application can register an interest in multiple queues. This is much simpler than having to poll a set of queues for the next available message.
- The application can choose to stop consuming from a queue at any time. When using MQGET with wait, the application is dependent upon a message arriving or the get-wait interval expiring.

On z/OS, asynchronous consume can be used in batch and CICS (CTS TS 3.2). Additionally, when a client application that has connected over a SVRCONN channel (with a non-zero SHARECNV channel attribute) issues an MQGET, the channel initiators uses asynchronous consume to get the message.

When the channel initiator uses asynchronous consume to get the message for a client application, put to waiting getter is not viable.

Asynchronous Consume in a Client Pub/Sub Environment

We have noticed when subscribing with clients over a server connection channel with a non-zero SHARECNV channel attribute which forces the use of asynchronous consume to drive messages onto the client, there is a delay in the messages being sent to the client. This performance delay is currently being investigated.

Performance Data

Memory Allocation

1. Queue manager restart time is reduced

By changing the queue managers' start-up code to allocate each bufferpools' storage in a single storage obtain call, there has been a reduction of more than 1 CPU second.

For example, a version 6.0 and a version 7.0 queue manager that have been configured with 4 bufferpools, 2 of which are 20,000 buffers, 1 is 50,000 buffers and the fourth is 99,000 buffers and both queue managers have the same objects defined, the version 6.0 queue manager used 2.6 CPU seconds to start. By contrast, the version 7.0 queue manager took 1.3 CPU seconds to start.

2. The queue managers' memory footprint is reduced.

With long-term storage protection set on by the WLM "storage critical" option, a Version 6.0 queue manager will page-in all of the bufferpool storage.
E.g. A "DISPLAY USAGE" command would return:

Page set	Buffer pool	Total pages	Unused pages	Persistent data pages	NonPersist data pages	Expansion	count
0	0	**20157**	19865	292	0	USER	0
1	1	**20157**	20157	0	0	USER	0
2	2	**20157**	20157	0	0	USER	0
3	3	**20157**	20157	0	0	USER	0
4	1	**20157**	20157	0	0	USER	0

Since all of these pages are in real storage, the queue manager storage footprint is 20157 * 5 = 100,785 pages.

By comparison, issuing the "DISPLAY USAGE" an equivalently configured version 7 queue manager would return:

Page set	Buffer pool	Total pages	Unused pages	Persistent data pages	NonPersist data pages	Expansion	count
0	0	20157	19865	**292**	**0**	USER	0
1	1	20157	20152	**0**	**0**	USER	0
2	2	20157	20157	**0**	**0**	USER	0
3	3	20157	20157	**0**	**0**	USER	0
4	1	20157	20157	**0**	**0**	USER	0

For the version 7 queue manager, only the used data pages are actually paged in, i.e. 292 pages from bufferpool 0. This gives a far smaller working footprint – 292 pages rather than 100,785 pages.

Effects on storage usage of defining additional objects

On version 6 any new objects defined will not require extra pages to be paged-in, provided that pageset expansion is not necessary since the bufferpools are held in real storage,. This means that the storage usage should remain constant irrespective of new objects defined.
Version 7 shows the cost of an object definition much more clearly as for each new object being defined, more pages will be required to be paged in. For example, having defined 1000 local queues, the "DISPLAY USAGE" command now returns:

Page set	Buffer pool	Total pages	Unused pages	Persistent data pages	NonPersist data pages	Expansion	count

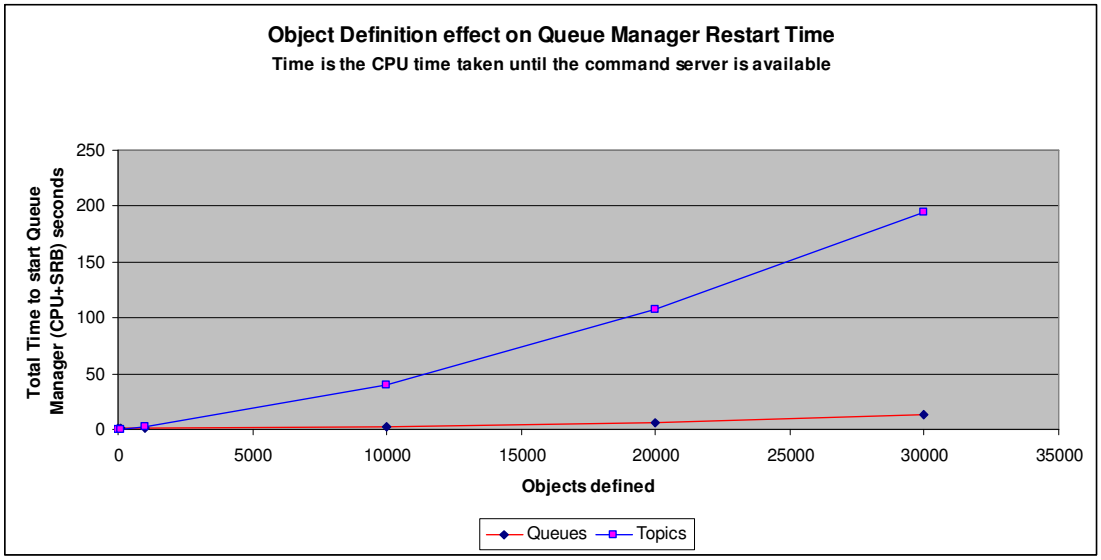
```
0       0       20157     19615      542          0 USER       0
1       1       20157     20152        5          0 USER       0
2       2       20157     20157        0          0 USER       0
3       3       20157     20157        0          0 USER       0
4       1       20157     20157        0          0 USER       0
```

The version 7 queue manager now has 547 real pages, an increase of 250 pages – all of which are in real storage. Since a page is 4K, the cost of defining a queue can be derived as 1K in bufferpool usage.

Administratively defined Topics

Typically a topic node is only defined when the tree or part of the tree requires specific non-default attributes. This should mean that there is a relatively small number of administratively defined topics.

The following chart gives an indication of how CPU-intensive a high number of topic objects can be and how they will affect the queue manager restart time. For comparison purposes, a similar number of local queues were defined.



As can be seen, the number of queues defined to a queue manager does not significantly affect the CPU time taken to restart the queue manager. However, a large number of topic objects can increase the restart time noticeably.

Message Properties

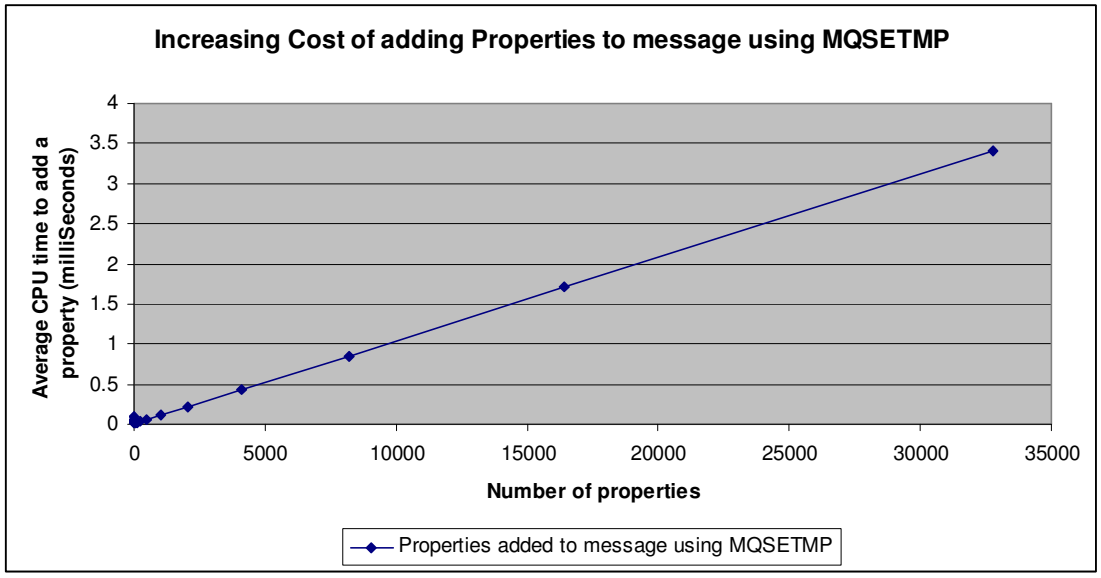
How many properties can be added to a message?

Whilst it is not envisaged that large numbers of properties will be added to a message, WebSphere MQ version 7.0 does support in excess of 30,000 properties, provided the destination queue supports the large message that will result from adding this number of message properties.

In the following measurement, a C application created a 100 byte message and added up to 32,768 message properties using the MQSETMP verb. The application then published the message to a topic that had one subscriber with PSPROP(NONE).

The application used a simple algorithm to specify the properties, where a 5 digit sequence number (numbered from '00001' to '32768') is concatenated to a 3 character string for the property name e.g. "IBM00001". The property value was a fixed length of 7 bytes.

When the message was consumed from the queue, the message buffer required to get the entire message was almost 1MB, which is a considerable overhead for a 100 byte message.



The cost of adding up to 64 properties is low, but once the number of properties exceeds 128, the cost begins to increase noticeably.

Subscriptions

The information reported in this section is based upon a Version 7.0 queue manager that has been created with 200,000 pages in PAGESET(2) – which is used by the durable subscription queue. Additionally the following changes are made to the queue manager:

- A small set of topic nodes is defined using MQSC.
- The queue manager is altered to have MAXHANDS(999999999)

An application is run to define a set of durable subscriptions to topics. The tree grows at an exponential rate and contains 78,000 subscriptions and can be defined as below:

Topic Tree structure

- Each tree has 1 admin node defined
- From this admin node, the application creates 5 children or subscription points. ("level 2")
- From each of the subscription points on level 2, a further 5 subscription points are defined ("level 3").

- This process is repeated until there are 5 levels, and the following table indicates how many subscription points there are on each level.

Level	Subscription Points
1	0
2	5
3	25
4	125
5	625

This gives a total of 780 subscription points in this topic tree. For each of these subscription points, the application creates 100 subscriptions, which results in 78,000 subscriptions for the topic tree.

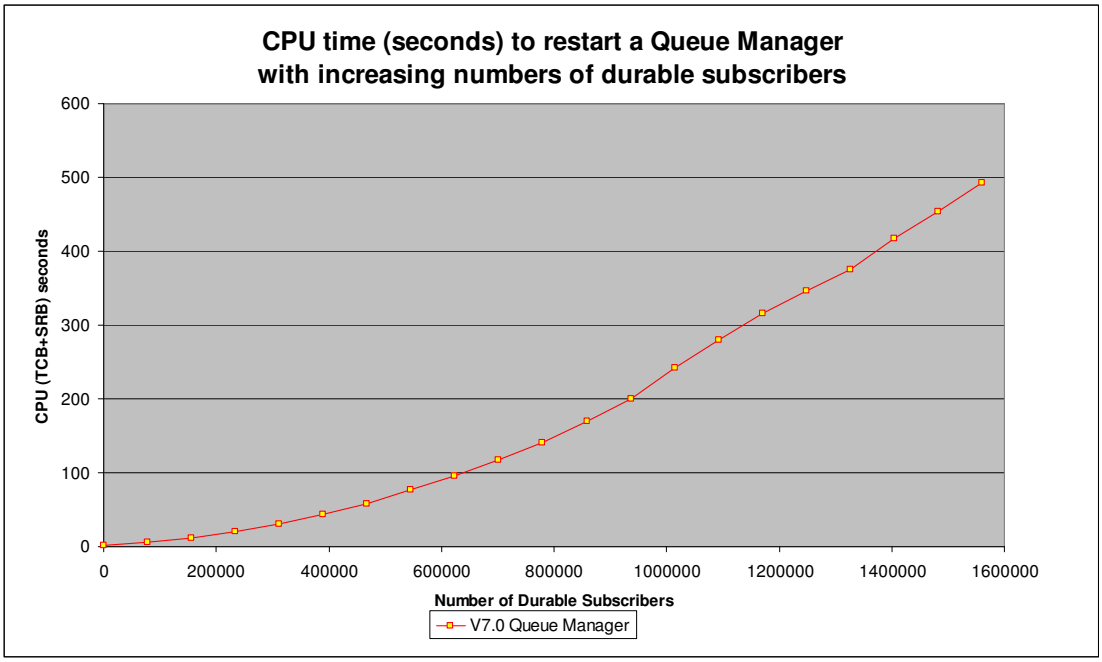
Following each set of subscriptions, the queue manager is restarted and a new set of subscriptions is defined – with a different admin node.

This process is repeated until there are 1.5 million durable subscriptions defined on the queue manager.

How long will my queue manager take to restart?

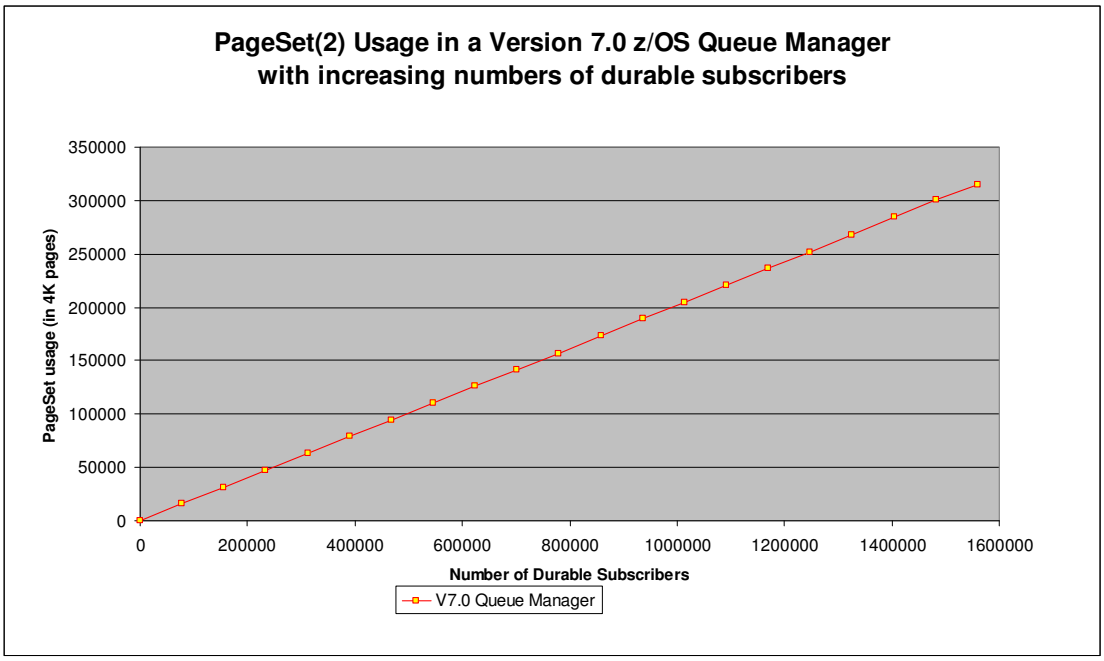
Using the process defined above, the queue manager has increments of 78,000 subscriptions defined between queue manager restarts.

The following diagram shows the CPU cost for each restart. This is the point where then queue manager is considered ready for business, i.e. all queue manager start up tasks are complete.



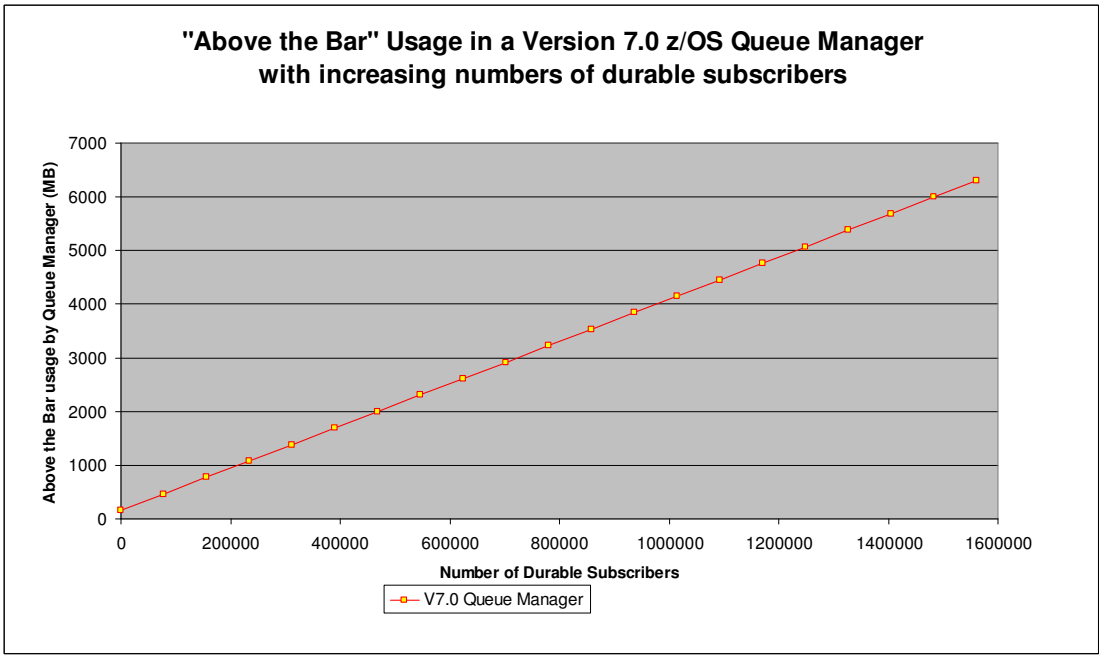
Pageset usage with Durable Subscriptions

The following chart shows the number of persistent data pages used in PAGESET(2) following queue manager restart with increasing numbers of subscriptions. This is for subscriptions where the maximum length of the MQSD SubName variable is 27 characters. As the length of the subscription name increases, fewer messages will be able to be compressed into the consolidated message.



Above the Bar usage with Durable Subscriptions

This chart show the number of megabytes of “above the bar” storage that the queue manager has allocated to retain the durable subscription information. Again the subscription name is a maximum of 27 characters.



Publish

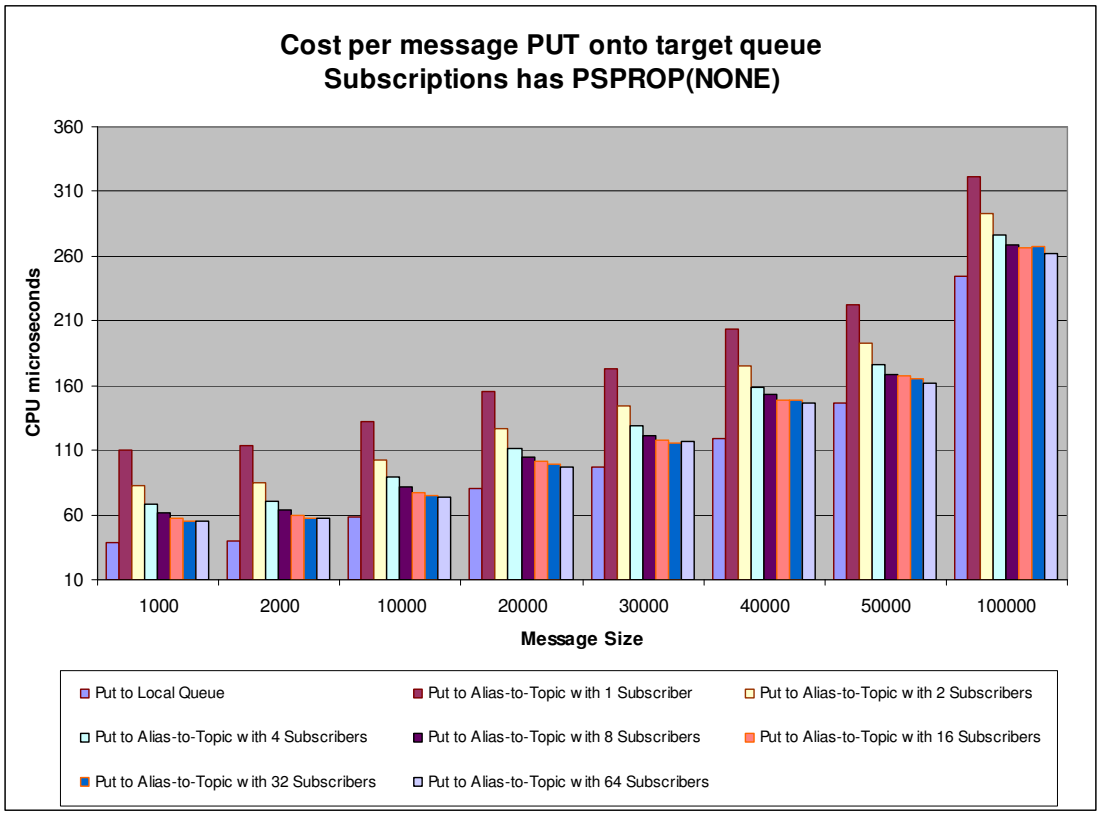
How does the cost of Put compare to Publish?

When an application publishes a message to a topic, the queue manager code will schedule a preemptible SRB and the MQPUT verb will not return until all subscriptions have been satisfied.

This can make the MQPUT run for an unexpectedly lengthy period of time. There are 2 ways in which this data can be represented:

1. How costly is a put to a queue compared to a put to each subscriber queue, i.e. if there are 64 subscribers to a topic, divide the total cost of the MQPUT to topic by 64.
2. How costly is the MQPUT to topic compared to issuing a separate MQPUT/MQPUT1 to each subscriber queue. This means that if a topic has 64 subscribers, the cost of the MQPUT to topic is compared against an application issuing 64 separate MQPUT verbs. Strictly speaking the application should issue 64 MQOPEN, 64 MQPUT and 64 MQCLOSE calls to simulate the publisher process but a well designed application would open and close the queues only once.

The chart below shows the cost per MQPUT to a queue compared to the cost of the MQPUT to topic divided by the number of subscribers.



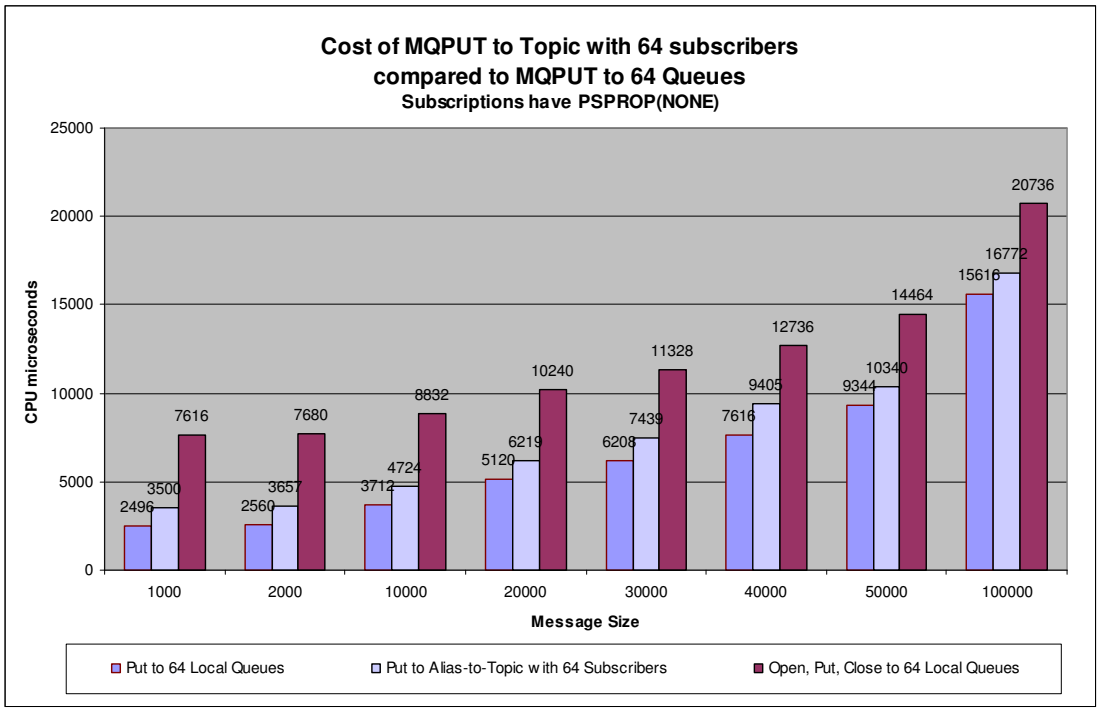
Note: Messages are non-persistent.

As can be seen from the above chart, as the number of subscribers to a topic increase, the cost of each put to the subscriber queue decreases.

Using different PSPROP options on the subscription made no noticeable difference to the costs of publishing.

Clearly there is a significant overhead in publishing when there are low numbers of subscribers but as the number of subscribers increase, the overheads diminish.

Comparing the total cost of MQPUT to topic with multiple subscribers against the cost of an application putting to queues gives the following chart:



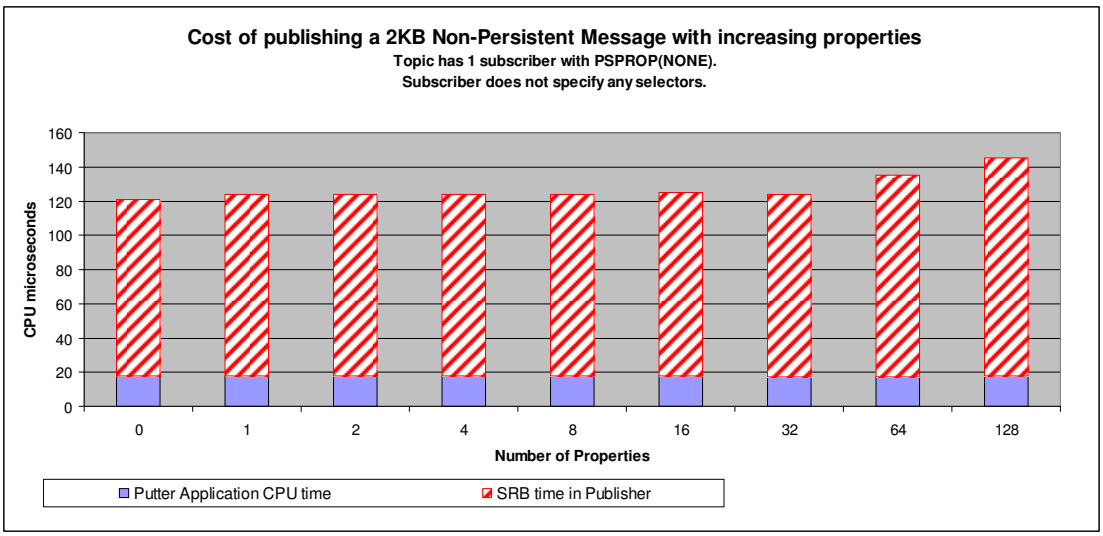
Note: Messages are non-persistent.

As can be seen from the above chart, an application that is considered to be badly behaved i.e. one that opens, puts and closes the queue repeatedly rather than open, put many and then close the queue, is more expensive than “publish once” to 64 subscribers. The open and close cost has been calculated at 80 microseconds.

Do message properties affect the cost of publish?

For a topic that has durable subscribers with no selection criteria on the subscription, there is a small overhead in the put of a message with properties. For a 2KB non-persistent message being published with 1 property, there is a 2% increase in the cost of the put over similar message with no properties.

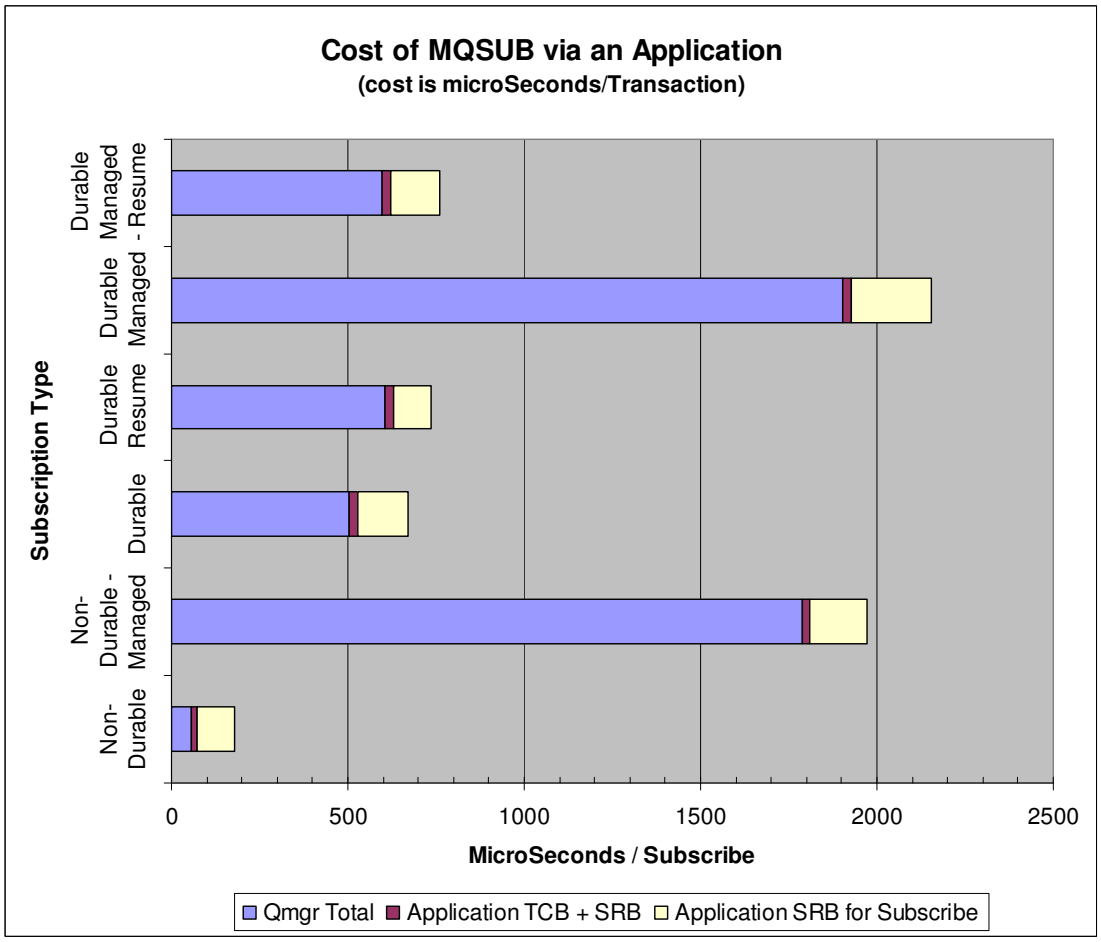
The following chart indicates the cost of publishing a message to a topic with increasing numbers of properties.



Subscribe

How much does an Application MQSUB cost?

The following chart gives an indication of the cost of an MQSUB for durable and non-durable subscriptions plus the overheads of using managed subscriptions.



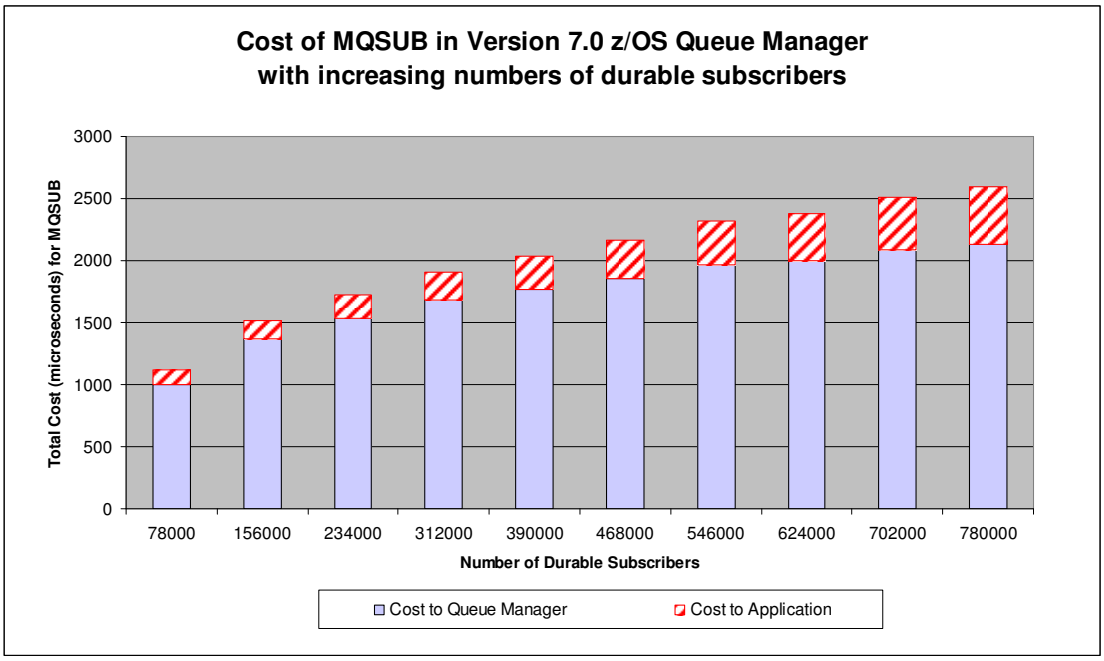
As can be seen from the chart, the managed subscriptions are significantly more expensive than the non-managed. The use of managed subscriptions does mean the administrator does not need to define the queues that will be used to host the published messages.

Comparing the cost of the durable managed subscription (2,154 microseconds) against the cost of the durable non-managed subscription (668 microseconds), would suggest that the overhead involved with managed subscription cost is high – but consider that for the non-managed durable subscription, the application has to specify a destination queue that has previously been opened. This destination queue will either have had to be pre-defined (at cost to the administrator) or will be a permanent dynamic temporary queue which will cost around 950 microseconds to create during the MQOPEN.

This means the cost of dynamically creating a queue and then using that queue as the destination for a durable subscription is 1.6 milliseconds compared to the non-durable managed subscription costing 2.1 milliseconds.

Cost of MQSUB as subscriptions increase

As the number of subscriptions defined to a queue manager increases, the cost of adding a new subscription will increase. The following chart gives an indication of the cost of a subscription.



Notes on chart:

1. The queue manager has been restarted between data points.
2. The cost shown is the average cost of a subscription until the total number of subscriptions matches the value in the x-axis, e.g. for the column labelled 78000, the average cost of each subscription was 1.12 milliseconds until the queue manager had 78000 durable subscriptions. When adding the next 78000 subscriptions the cost of the subscription increased to 1.5 milliseconds.

Publish/Subscribe

Publishing when subscribers specify message properties

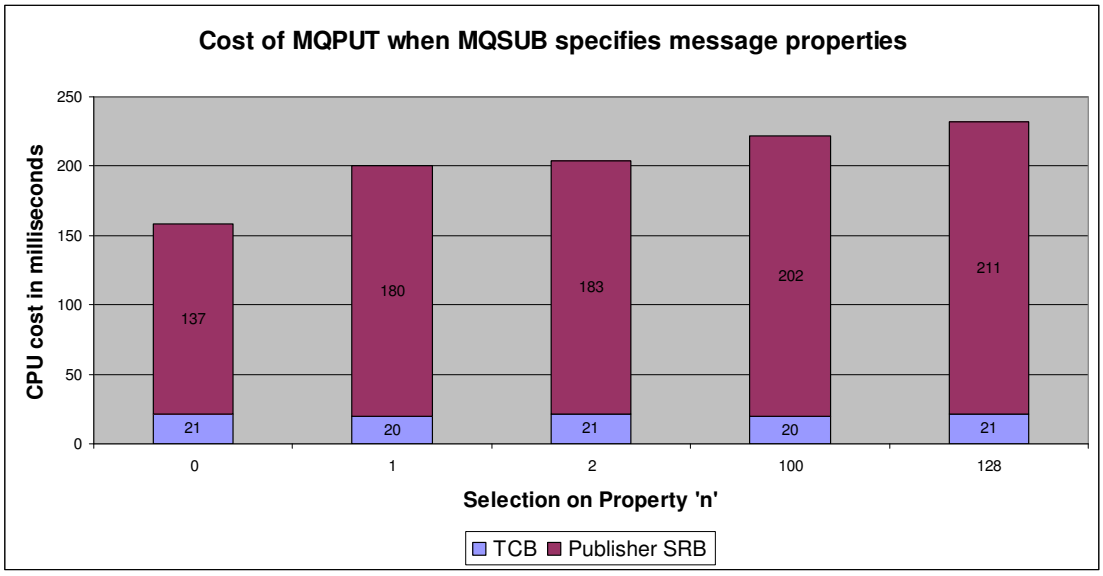
To show how specifying message properties at subscription time can affect the cost of publishing a message, the following scenario was measured:

An application was run to generate 2KB non-persistent messages with 128 properties. Another application was run to subscribe to a topic using a number of different criteria:

- Subscribe with 0 selectors
- Subscribe with 1 selector – on property 1
- Subscribe with 1 selector – on property 2
- Subscribe with 1 selector – on property 100
- Subscribe with 1 selector – on property 128.

For all measurements in this scenario, a pre-defined queue was used as the destination for the published message.

The following chart shows the cost of the MQPUT by the publishing application. The cost of the MQGET is consistently around 210 microseconds for each 2KB message. When the put to waiting getter[3] criteria is satisfied, the cost of the MQGET dropped to approximately 20 microseconds for each message.



The x axis shows the property number that is being selected on, i.e. where the x axis has 0, the subscription has specified no selection criteria, whereas for the value of 100, the subscriber has selected on property 100.

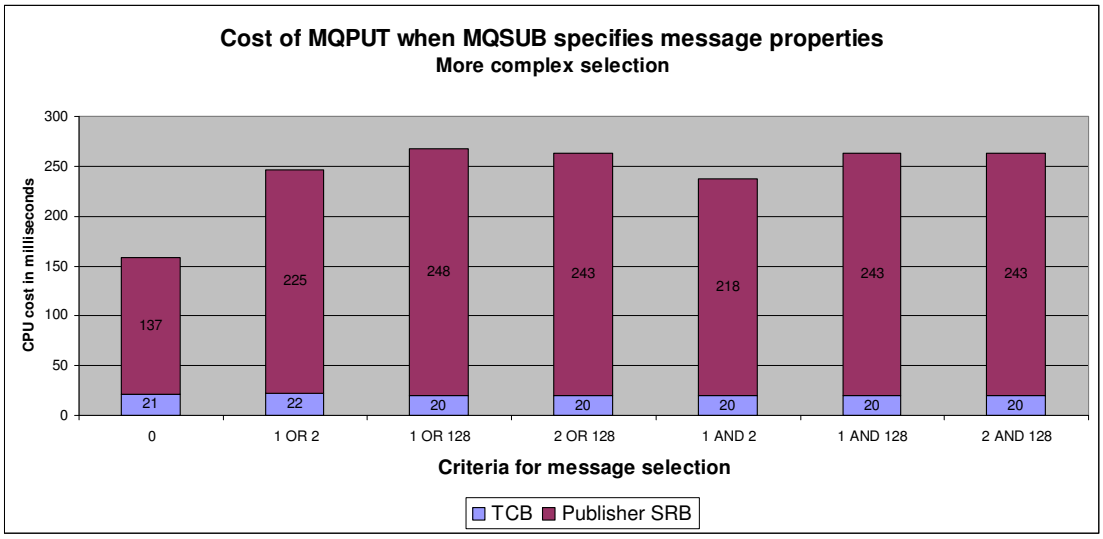
The CPU costs shown as the y axis are for the queue manager and the application combined.

[3] Put to Waiting Getter was introduced in version 6.0 of WebSphere MQ for z/OS and allows an out-of-syncpoint non-persistent message to be MQPUT directly to a waiting out-of-syncpoint MQGETter, rather than being placed onto a queue and then read from the queue.

Using message properties to select messages can allow more complex selection criteria to be specified, by using "AND" or "OR" to create compound selection statements. To indicate the cost of using these criteria, further measurements were run using the following scenario:

An application was run to generate 2KB non-persistent messages with 128 properties. Another application was run to subscribe to a topic using a number of different criteria:

- Subscribe with 0 selectors (as a baseline)
- Subscribe with 2 selectors – on property 1 OR property 2
- Subscribe with 2 selectors – on property 1 OR property 128
- Subscribe with 2 selectors – on property 2 OR property 128
- Subscribe with 2 selectors – on property 1 AND property 2
- Subscribe with 2 selectors – on property 1 AND property 128
- Subscribe with 2 selectors – on property 2 AND property 128



As can be seen when comparing the 2 charts, the cost of selecting multiple properties is noticeable even when the selection is on the earlier properties. When selecting on property one, the cost was 200 microseconds but when selecting on property one or property two, the cost increased by a further 47 microseconds.

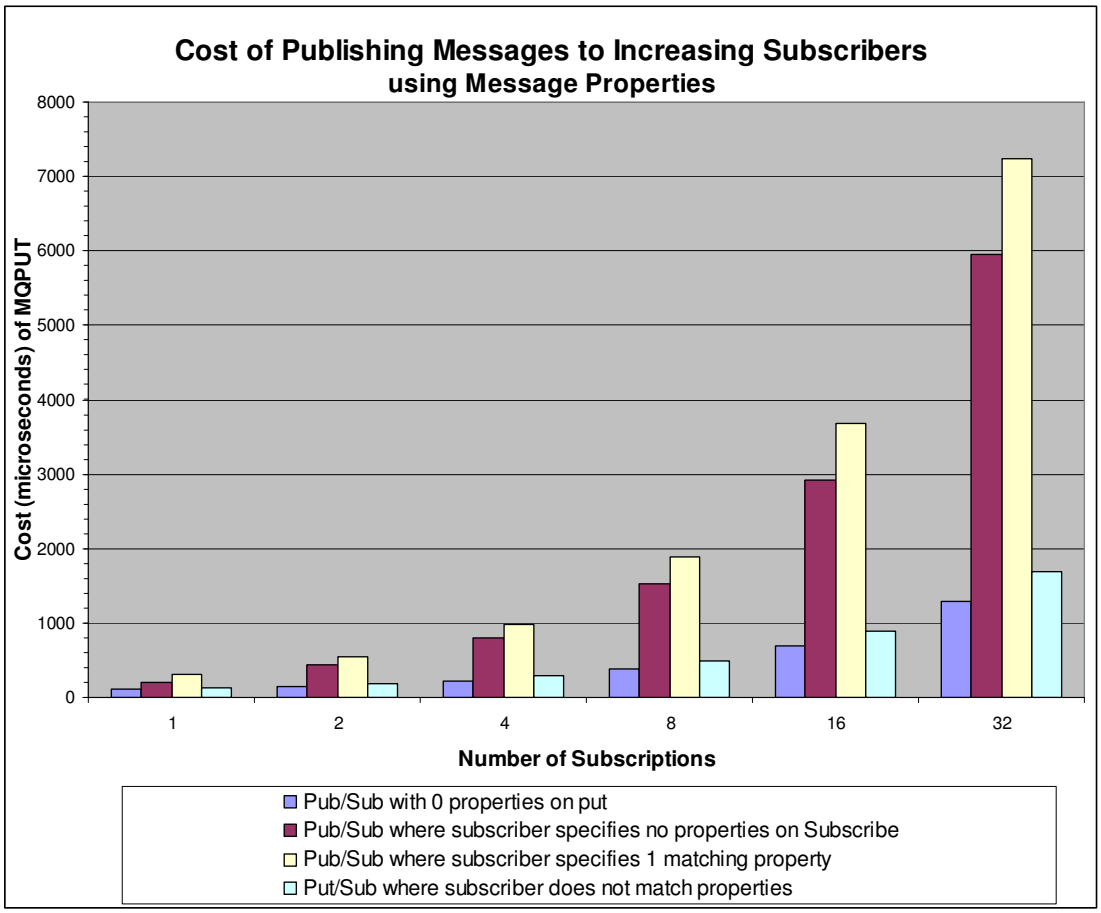
Publish/Subscribe on a Local Queue Manager

To illustrate how publishing messages with properties scales with increasing numbers of subscribers, the following scenario was measured:

- An increasing number of subscriber applications were run. The subscriber would either specify
  - 0 properties on the subscribe
  - 1 property that would match the publisher
  - 1 property on the subscribe that did not match the publisher
- Publisher application(s) were run with
  - 0 properties

  - 128 properties
- A 2KB non-persistent message would be used
- Subscriptions would be non-durable and would use a temporary dynamic queue

As a reference, publishing a message for this configuration where there were no properties on message being put and no subscribers to the topic, saw a cost to the MQPUT of 55 microseconds. When there were 128 properties associated with the message being put and still no subscribers to the topic, the cost of the MQPUT increased to 68 microseconds.



Notes on chart:

- The messages being published have 128 properties associated unless otherwise stated.

If message properties are not required on the subscriber, it is significantly cheaper for the publisher application to publish without properties.

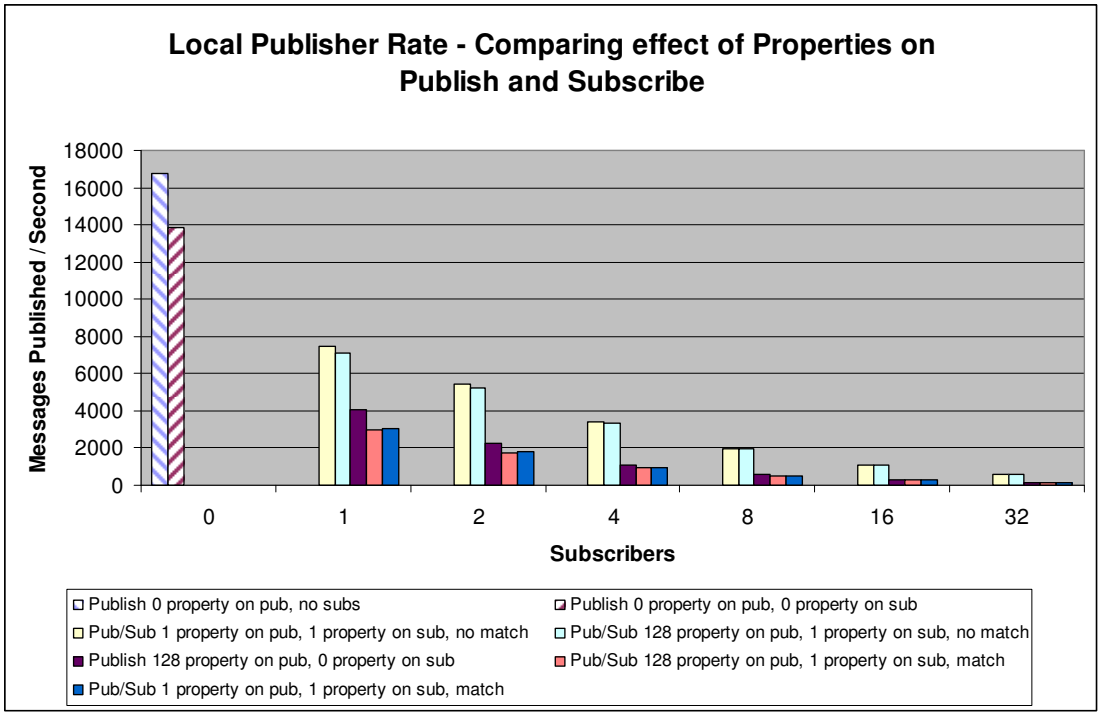
If a topic has subscribers but the message selectors specified do not match the message properties on the published message, the publisher will still incur a cost when attempting to match the properties on the message with those of the subscribers.

The following chart shows the number of MQPUTs completed per second for a batch application putting a 2KB non-persistent message where there are increasing numbers of non-durable subscribers. There are 7 scenarios measured:

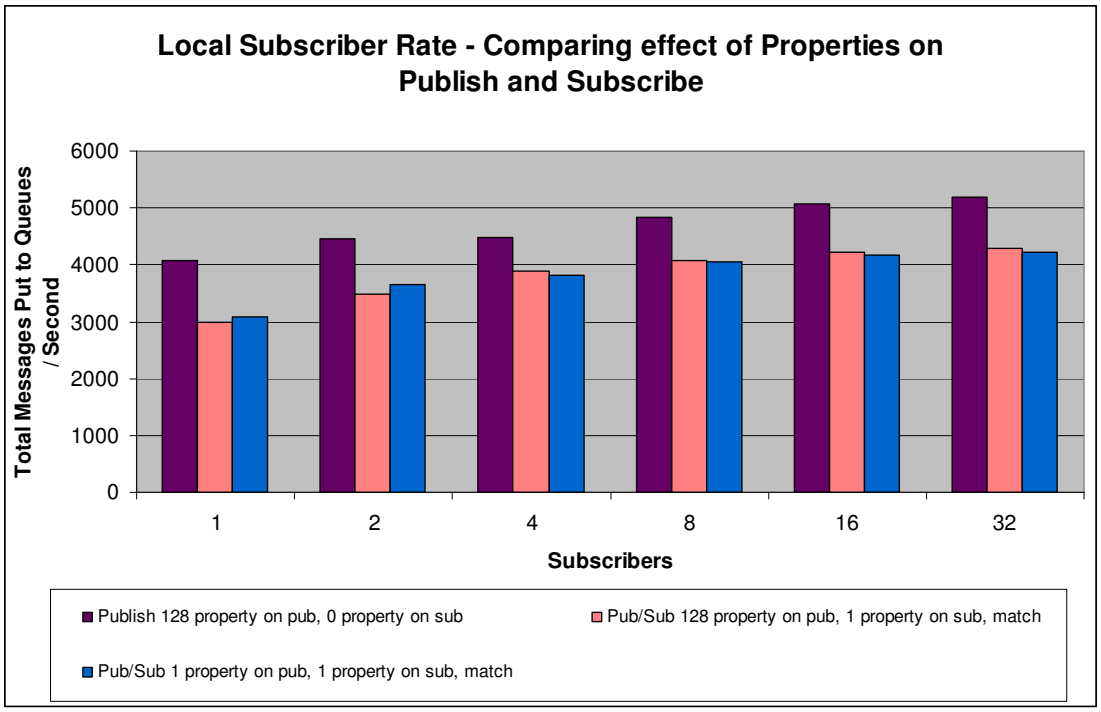
1. Publish message with no properties, where there are no subscribers to the topic.

2. Publish message with 128 properties, where there are no subscribers to the topic.
3. Publish message with 1 property, where subscriber has specified 1 selector that does not match the publishers' property.
4. Publish message with 128 properties, where subscriber has specified 1 selector that does not match the publishers' property.
5. Publish message with 128 properties, where subscriber does not specify any selectors
6. Publish message with 1 property, where subscriber specifies 1 selector that matches the publishers' property.
7. Publish message with 128 properties, where subscriber specifies 1 selector that matches the publishers' property.

Only scenarios 5, 6 and 7 see messages received by the subscriber applications.



The following chart shows the total rate per second that the messages get to the subscribers' queues. The same scenarios were run as in the previous chart, but only the scenarios that result in messages getting to queues are displayed. The numbers are calculated by multiplying MQPUTs per second by the number of subscribers.



Benefits of using message properties
If a subscribing application will require every message that is published to a topic, message properties may be an unnecessary overhead, both on publish and subscribe.

However, if a subscriber application only requires for example 1 in 4 messages that get published to the topic, it may be cheaper or easier for the application to use message properties to filter out the unwanted messages.

Earlier in this section, the cost of the MQGET for a 2KB non-persistent message was established to be 20 microseconds when put to waiting getter was viable and 210 microseconds when it was not.

The following table shows a comparison of the expected costs comparing when the subscriber application has to filter out unwanted messages rather than relying on selecting on message property in the MQSUB verb.

The cost shown for the MQPUT is for a message with 128 properties. Measurements with 1 property show a negligible cost saving on publish where the subscriber matches on the message property. Where the subscriber does not match the publishers' message property, a 2KB non-persistent message published with 1 property is 2% cheaper than a message published with 128 properties.

This following table is based on a subscriber only being interested in 1 in every 4 messages that is published.

Cost is in microseconds (uS).

	Subscribing Application provides filtering logic				MQ Message Properties and Selectors filter undesired messages			
Subscribers	**1**	**1**	**32**	**32**	**1**	**1**	**32**	**32**
Publisher of the 1 desired message								
MQSETMP (128 properties)	N/A	N/A	N/A	N/A	25	25	25	25
MQPUT	104	104	1294	1294	312	312	7239	7239
Publisher of the 3 undesired messages								
MQSETMP (128 properties)	N/A	N/A	N/A	N/A	75 (25*3)	75 (25*3)	75 (25*3)	75 (25*3)
MQPUT	312 (104*3)	312 (104*3)	3882 (1294*3)	3882 (1294*3)	396 (132*3)	396 (132*3)	5064 (1688*3)	5064 (1688*3)
Subscriber								
MQGET put to waiting getter viable	80 (20 *4)		2560 (20*4*32)		20[4]		640 (20*32)	
MQGET Put to waiting getter not viable		840 (210*4)		26680 (210*4*32)		210 (210*1)		6720 (210*32)
TOTAL Cost	**496**	**1256**	**7736**	**32056**	**828**	**1018**	**13043**	**19123**
Cost / Subscriber	**496**	**1256**	**242**	**1002**	**828**	**1018**	**407**	**598**

For the cases where the subscribing application provides the filtering logic, there will be additional cost in each subscribing application to parse the message data to determine whether the message is valid.

As can be seen from the above table, even when put to waiting getter is viable, using message properties to filter out undesired publications can save processing cost when only 1 in 4 messages is appropriate to a subscribing application.

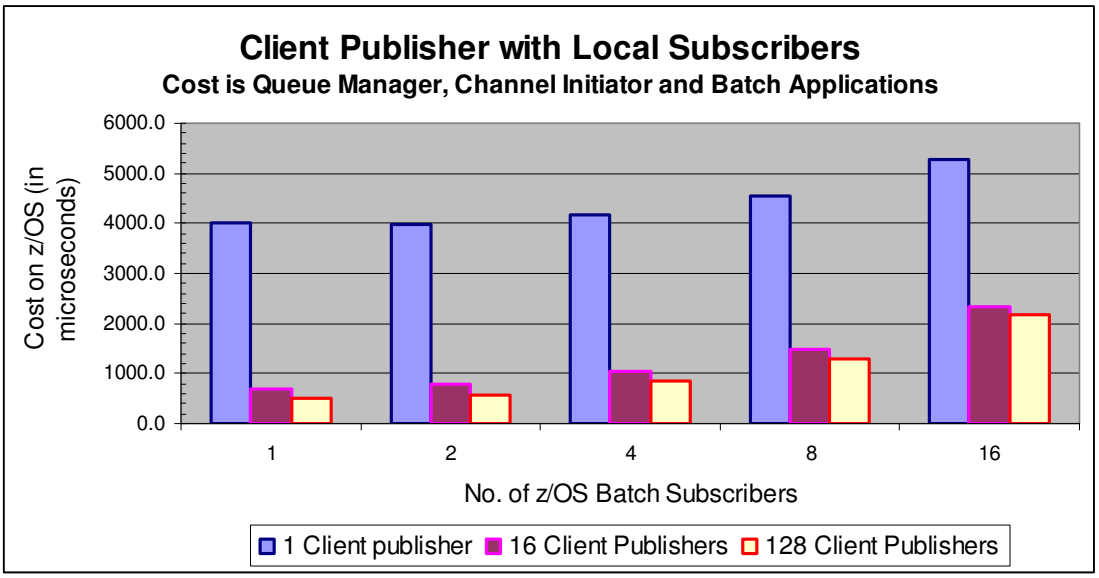
Client Publishers to Local Subscribers

The following scenario shows the cost of using client applications connected over a server-connection channel to publish when the subscribers to the topic are hosted on z/OS.

- Each publisher will publish a 2KB non-persistent message every 5 seconds to the same topic.
- An increasing number of subscribers to the topic are measured. Each subscriber is using a separate permanent dynamic queue.
- An increasing number of publishers are measured.

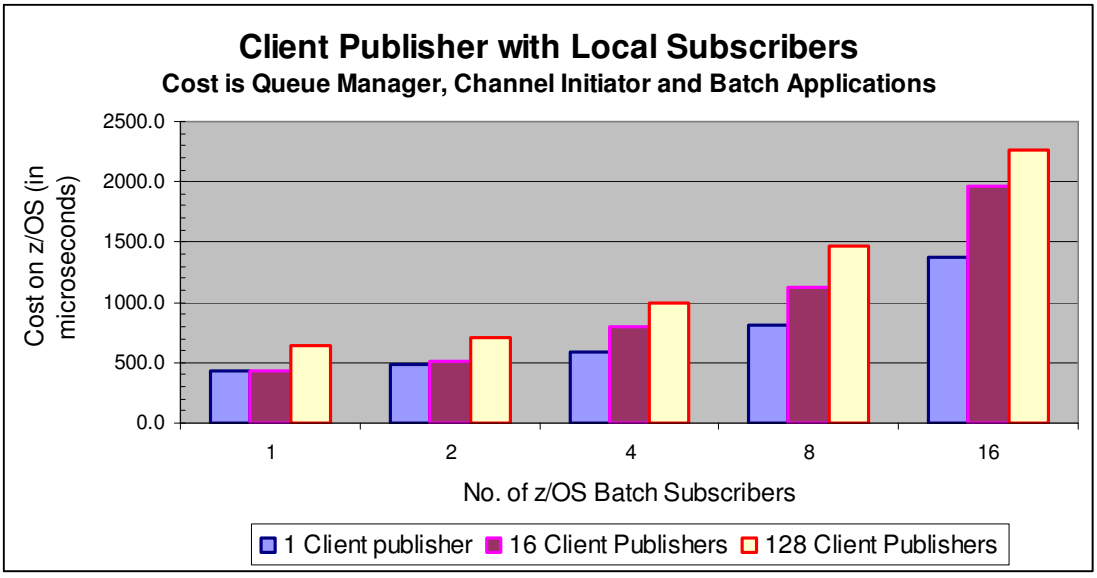
[4] The 3 messages that the publisher puts that do not match the subscribers selection criteria are not gotten, so the subscribers' MQGET only has to account for the single message received.

- The cost is by transaction and includes the cost incurred by the queue manager, channel initiator and batch application(s) getting the published message.
- A transaction is defined as "*a client publisher putting the message and all subscribers getting their copy of the published message*". This means that when there is 1 publisher with 16 subscribers, the cost will include the cost of publish to the channel initiator, the replication of the message to the 16 destinations, and the cost of each of those 16 subscribers actually getting the message.
- Over a 4 week period, an idle queue manager and channel initiator was seen to cost 33.7 CPU seconds per day (queue manager was 24 CPU seconds and channel initiator was 9.7 CPU seconds). This equates to 390 CPU microseconds per second. This "idle cost" has been deducted from the cost per transaction.



The chart confirms that for each number of publishers, as the number of subscribers increase, the cost per transaction also increases.
The more interesting data appears to show that the cost per transaction is higher when there are fewer publishers. This highlights the cost of the scavenger process running once the messages are gotten, i.e. when 1 publisher is putting a message every 5 seconds and the messages are gotten as they appear on the queues, the scavenger process will run 12 times in 60 seconds and each transaction will incur the full cost of the scavenger process from start up to close down, whereas when 16 publishers are running at the same rate, there are 16 times as many transactions per minute. As a result the cost of processing a message is lessened as the scavenger overheads are reduced by a factor of 16 as it will only run when the messages have been removed from the queue.

The followed scenario is similar to the previous one except that the focus is on the number of messages that the subscriber receives in a fixed period, rather than the number of messages published at a fixed rate. Unlike the previous measurement, where the publishers were putting 1 message every 5 seconds independently of the number of subscribers, this measurement ensures that each subscriber gets a fixed number of messages per hour (46,000). This ensures that the scavenger overheads are similar in all cases.



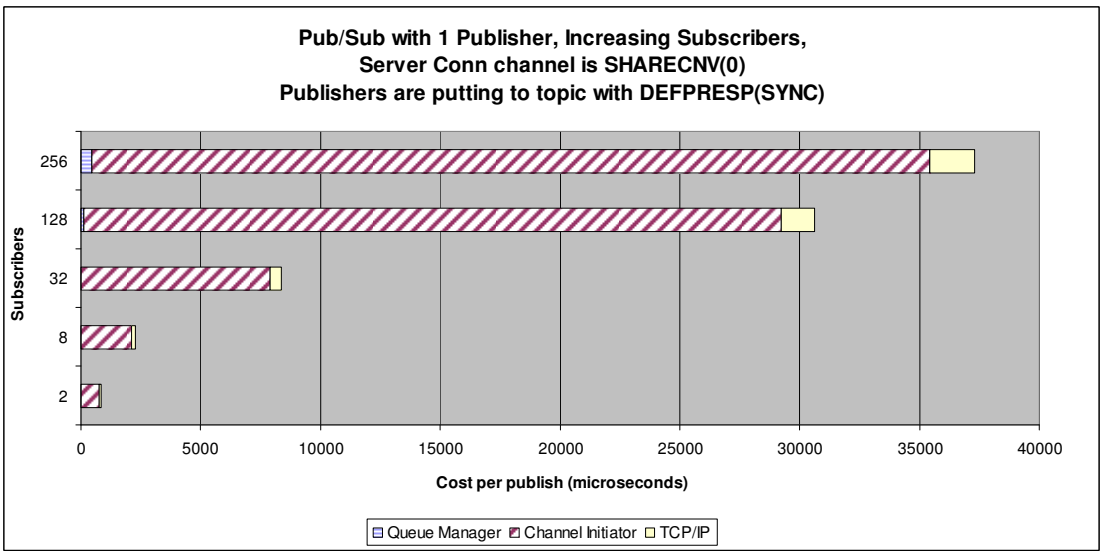
In this measurement, the cost per transaction does increase as the number of publishing applications increase. It also shows the cost of running 1 publisher is less than running 16 or 128 publishers.

Local Publish to Client Subscribers

When running a local publisher with client subscribers, the local publisher will be able to publish messages at a far faster rate than the subscribers will be able to get or be given the messages. For large numbers of locally published messages that are published at high rates, this can lead to deep queues, and in the case of non-persistent messages can result in messages being written to disk, which will slow the rate more.

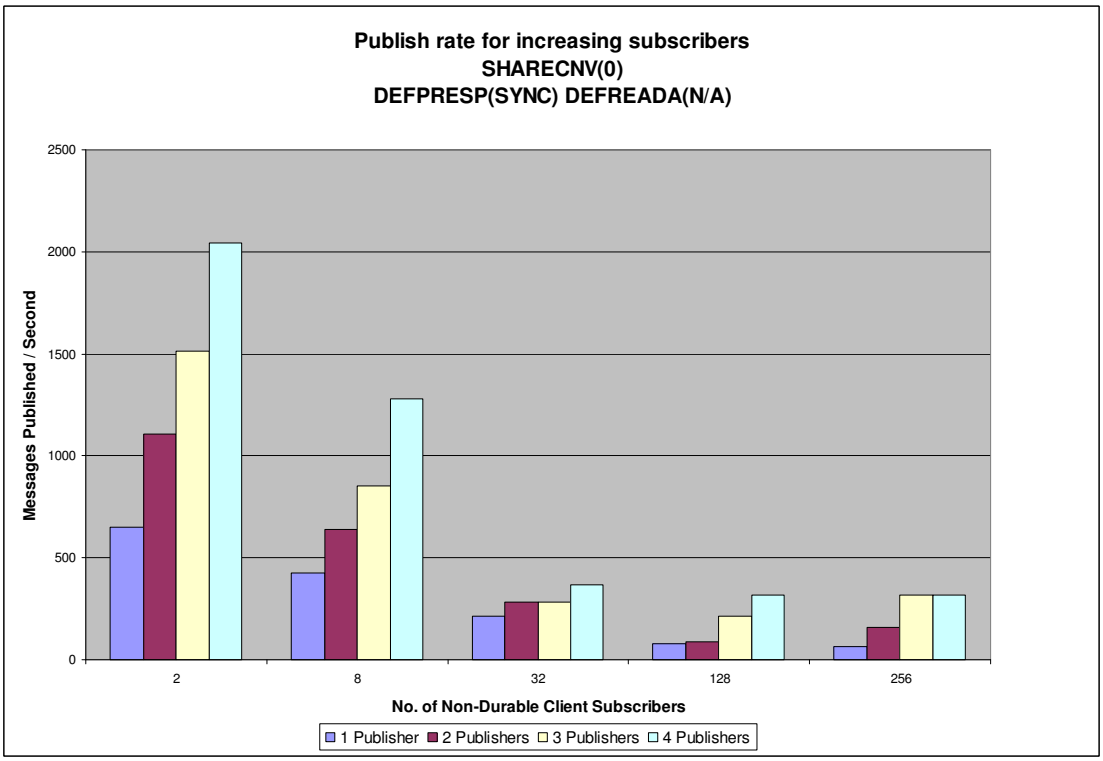
Client Publish to Client Subscribers

The first chart below shows the breakdown of the cost accounted to the resources involved when a client application publishes to a topic on a z/OS queue manager that has increasing numbers of subscribers. The publishing application(s) are publishing as fast as possible.
As can clearly be seen and was stated earlier in this document, the majority of the cost involved is within the channel initiator address space.
The chart shows the cost of a single client publisher with increasing numbers of non-durable client subscribers with no selection criteria. The publisher puts a 2KB non-persistent message.

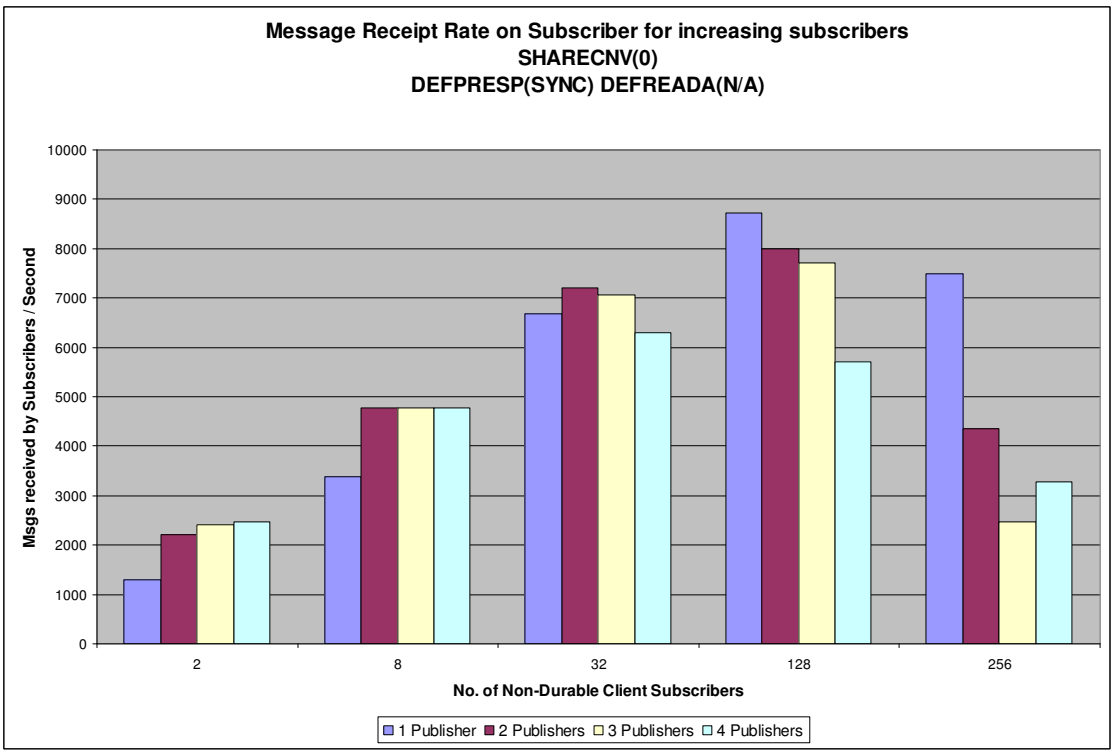


In the above scenario, running with DEFPRESP(ASYNC) makes little difference to cost as the majority of the work involves the pub/sub fan out to multiple subscribers and the retrieval of the published messages by the client subscribers.

The following chart indicates the rate at which the publishers are able to put messages to the topic.

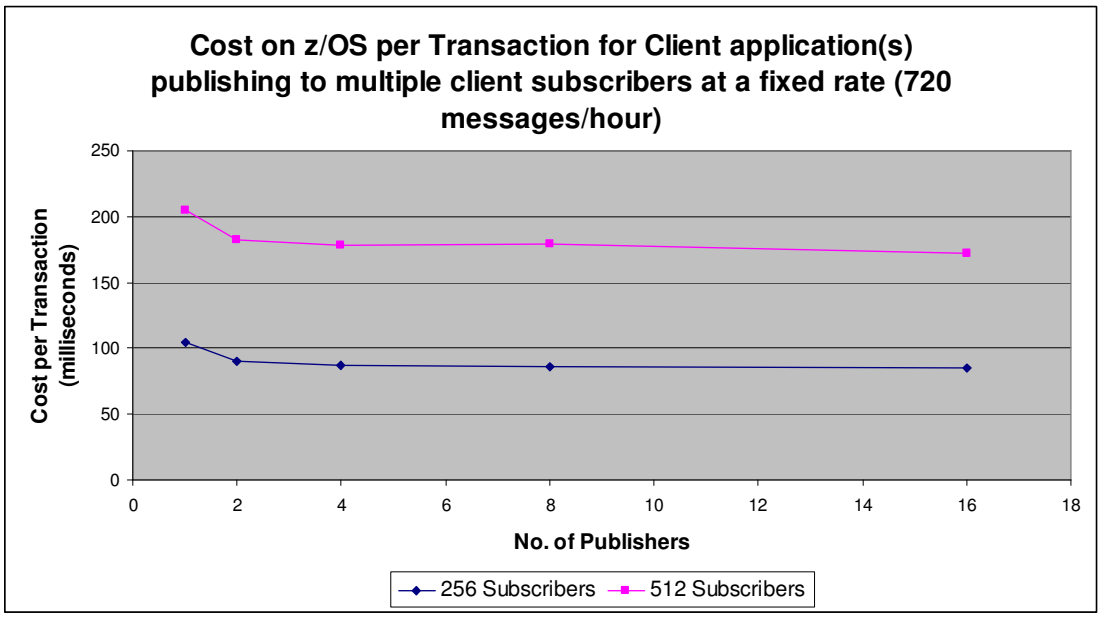


As a contrast, the next chart shows the rate the messages are received by all the client subscribers.



The client subscription rate drops off as the number of messages being written increases, partly due as gets are slower than puts, so the queue depths begin to build up. As a result messages get written to disk and the get becomes slower still as they need to be read from disk – and when there are large numbers of subscribers, there are a large number of reads from disk. Larger bufferpools will reduce this slowdown up to a point. Realistically, publisher application(s) are unlikely to be publishing as fast as they can put a message – instead some business logic is likely to be involved.

The next scenario is where the numbers of subscribers is fixed at 256 and 512. The number of publishers varies, from 1 to 8. Each publisher will attempt to publish 1 message to a topic every 5 seconds.



Notes on chart:

- In this measurement, a transaction is defined as the publisher putting the message plus the cost of delivering the message to all subscribers.
- The costs shown are the total costs incurred by the queue manager, channel initiator and TCP/IP address spaces.
- The chart shows the cost of publishing to a topic decreases slightly as the number of publishing applications increase in a throttled-back environment, i.e. the publisher is not publishing as fast as it can.

The chart also shows that by doubling the subscribers, the total cost of publish/subscribe has doubled. This does not mean that the cost of publish has doubled, as this total includes sending messages to an additional 256 subscribing applications.

Server-Connection Attribute SHARECNV()

The maximum number of channels is limited by

- Channel initiator virtual storage in the extended private region (EPVT), which includes all of the channel types, including CHLTYPE(SVRCONN) channels.
- Possibly, by achievable channel start (or restart after failure) and stop rates and costs.

Under WebSphere MQ version 6, every non-SSL channel uses about 140KB and every SSL channel about 170KB of extended private region in the channel initiator address space. Storage usage is increased if messages larger than 32KB are being transmitted.

Both the SHARECNV server-connection channel attribute and the channel initiator buffer pool functionality can affect the maximum number of channels that a channel initiator can support.

With the introduction of channel initiator buffer pools, the size of the message directly affects the memory footprint.

The following table shows the cost in memory of each new SVRCONN channel being run. The cost is in Kb.

SHARECNV value	1K Messages	10K Messages	32K Messages	64K Messages
0	90	106	170	202
1	234	228	351	418
10[5] Value in ( ) is footprint per connected client.	514 (51)	546 (55)	947 (95)	1327 (132)

[5] Channel is defined with SHARECNV(10) and has 10 shared conversations using the same server-connection channel instance as can be seen from the "DISPLAY CHS(*) CURSHCNV" command.

On a system that has an EPVT size of 1.6GB, this means that running server-connection type channels with SHARECNV(0) with 1KB messages, the maximum number of clients that can be connected should be able to reach the WebSphere MQ defined limit for MAXCHL of 9,999.

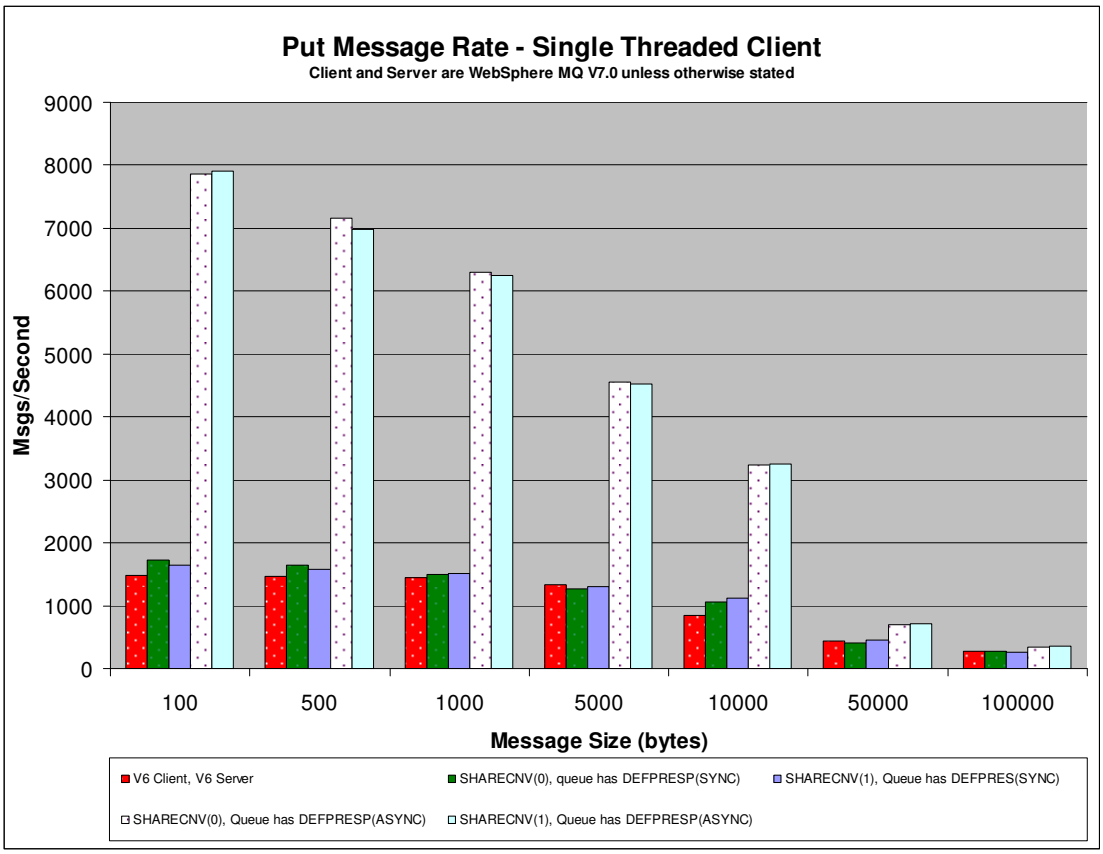
Compared to a server-connection channel with a SHARECNV value of 1 or more, where there are no shared conversations, the limit is much lower at 7,000 clients.

However, if shared conversations are being used on a server-connection channel with a SHARECNV value of 10, it is possible to have in excess of 25,000 clients connected concurrently.

Asynchronous Put

The rate at which a client can put non-persistent messages is significantly improved using the DEFPRESP(ASYNC) queue option where the client application set the MQPMO option “MQPMO_RESPONSE_AS_Q_DEF”.

In a queuing mode, i.e. the client application is just putting messages and not getting replies, a single threaded client application can be seen to perform as below:



In the above example, the client application is putting the same message many times and performs no business logic between MQPUTs.

Whilst the above message rates are for local queues over local SVRCONN channels, similar throughput has been seen for shared queues and shared SVRCONN channels.

How much cheaper are asynchronous puts?

If your messaging requirements allow it, running with asynchronous puts from a client application can reduce the cost of the put on the z/OS queue manager and channel initiator by between 18% and 55% for a messages ranging in sizes from 100,000 to 1000 bytes.

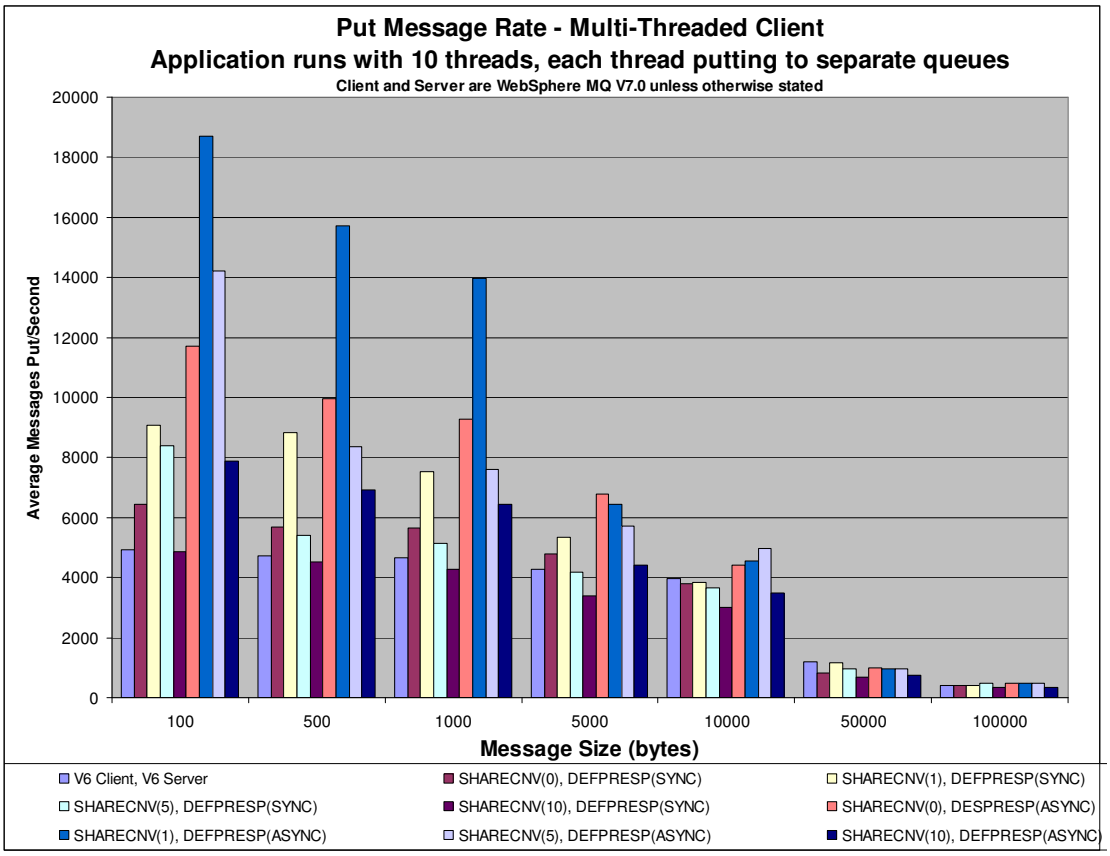
Shared conversations with asynchronous puts

By setting the server connection channel attribute “SHARECNV” to a value greater than one it is possible to run multiple conversations down a single channel instance.

Where the multi-threaded application is sending large amount of data to a z/OS queue manager, it is advisable to run with multiple channel instances rather than pushing all the data down a single channel instance.
Conversely where the multi-threaded client application is sending messages less frequently, it may be beneficial to have a higher value in the SHARECNV attribute.

In the following example, there is a single client application that runs 10 threads, each of which is putting to a separate z/OS queue and each client-thread is putting messages to their queue as fast as it can.

SHARECNV value	SVRCONN channels instances
0	10
1	10
5	2
10	1



With the larger messages, the various SHARECNV and DEFPRESP configurations are not being exploited as the application is attempting to send a volume of data close to the limits of the TCP/IP network used in this measurement – around 50MB/Second.

The smaller messages are not being sent fast enough to approach the network limits. Taking the put message rate for the 1000 byte messages and calculating the volume of data being transmitted over the network gives the following table:

WMQ Version	SHARECNV value	SVRCONN channels instances	DEFPRESP	MB/Sec (allowing 500 bytes for MQ Headers)
6.0	N/A	10	N/A	6.7
7.0	0	10	SYNC	8.0
7.0	1	10	SYNC	10.8
7.0	5	5	SYNC	7.3
7.0	10	1	SYNC	6.1
7.0	0	10	ASYNC	13.3
7.0	1	10	ASYNC	20
7.0	5	5	ASYNC	10.9
7.0	10	1	ASYNC	9.2

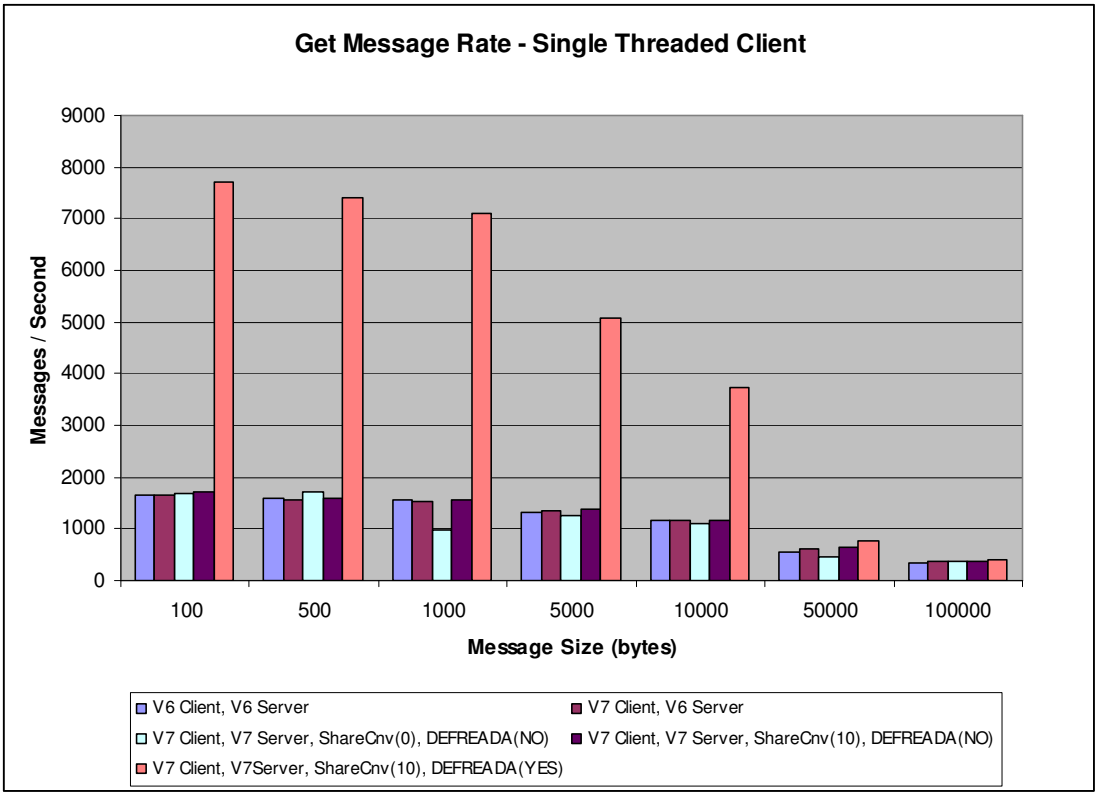
When sending large volumes of non-persistent messages from a multi-threaded client application, there are clearly benefits in not sharing conversations in a channel instance, since for the SHARECNV(1) measurement, there is a 20MB/sec throughput rate, which drops significantly when the SHARECNV() value is increased.

However, if the application is not flooding the network, the benefits of running shared conversations over a channel instance can be seen on the channel initiator as there is a lower memory usage per conversation in a shared channel instance.

Read-Ahead DEFREADA(YES | NO)

The rate at which a client can get non-persistent messages is significantly improved using the DEFREADA(YES) queue option.

In a queuing mode, i.e. the client application is just getting messages from a queue and is not putting message, a single threaded client application can be seen to perform as below:



In the above example, the queues are pre-filled and the client application is getting the next available message from the queue. Once the client application has got one message, it performs no business logic before issuing the next MQGET.

Whilst the above message rates are for local queues over local SVRCONN channels, similar throughput has been seen for shared queues and shared SVRCONN channels.

Are Messages on a DEFREADA(YES) Queue cheaper to consume?

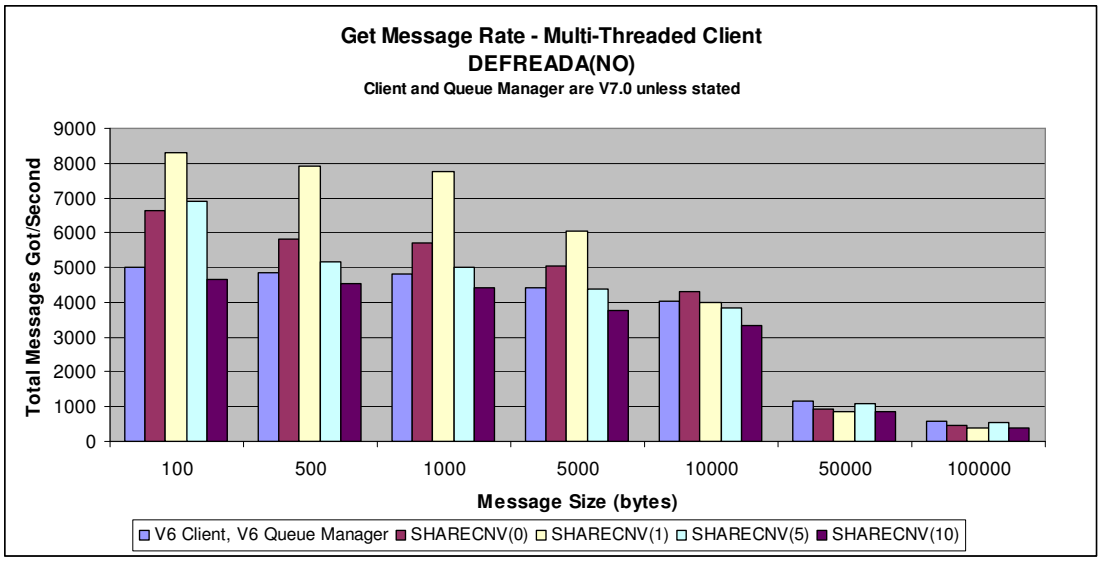
Provided the messaging requirements allow it, the cost to the queue manager and channel initiator on z/OS of a client getting a message from a queue that has the DEFREADA(YES) attribute set can reduce the cost of the MQGET on the queue manager and channel initiator by between 55% for messages of size 1KB to 12% for messages of size 50KB.

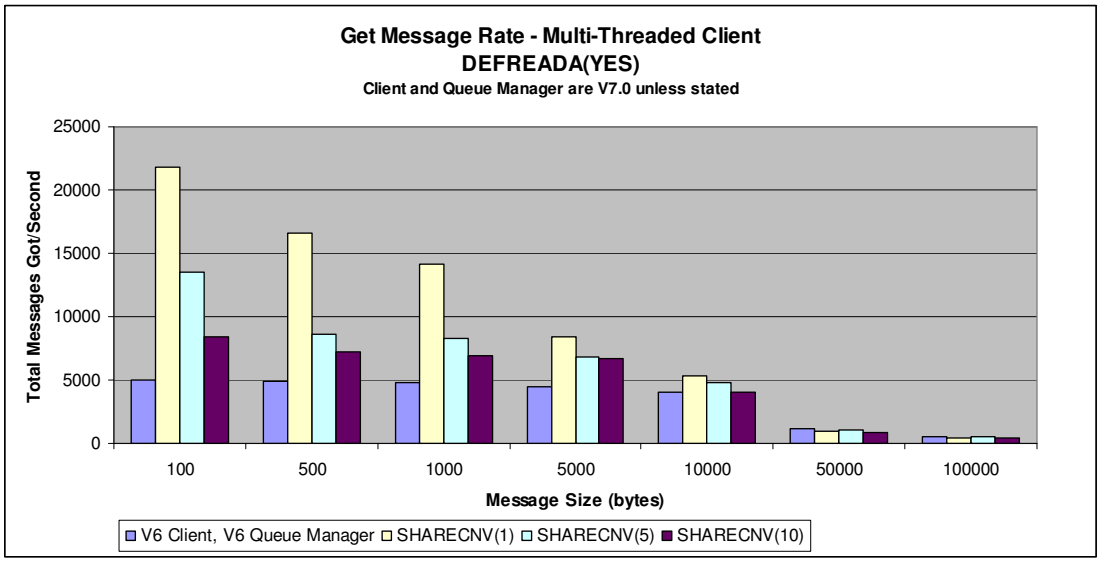
Shared Conversations with Read-ahead

By setting the server connection channel attribute “SHARECNV” to a value greater than one it is possible to run multiple conversations down a single channel instance.

In the following example, there is a single client application that runs 10 threads, each one is getting messages from a separate z/OS queue and each client-thread is getting messages from their queue as fast as it can.

SHARECNV value	SVRCONN channels instances
0	10
1	10
5	2
10	1





Note: In the above chart, there is no measurement for SHARECNV(0) as read-ahead is not available in that configuration.

Both charts indicate that Version 7.0 is able to get messages at a faster rate than Version 6.0. The optimum rate can be achieved by using SHARECNV(non-zero) configured channels, where conversations are not shared. However the channel footprint, as indicated in section "Server-Connection Attribute SHARECNV()" may preclude the use of SHARECNV(non-zero) server connection channels unless the installation is able to share conversations over those channel instances.

Where it is required to sustain a high message get rate of non-persistent messages, the DEFREADA(YES) queue option will provide a visible benefit, so it could be beneficial to have multiple server-connection channels defined, some with the SHARECNV attribute set to 1 and restrict use of this channel to a relatively low number of clients.

Asynchronous Consume

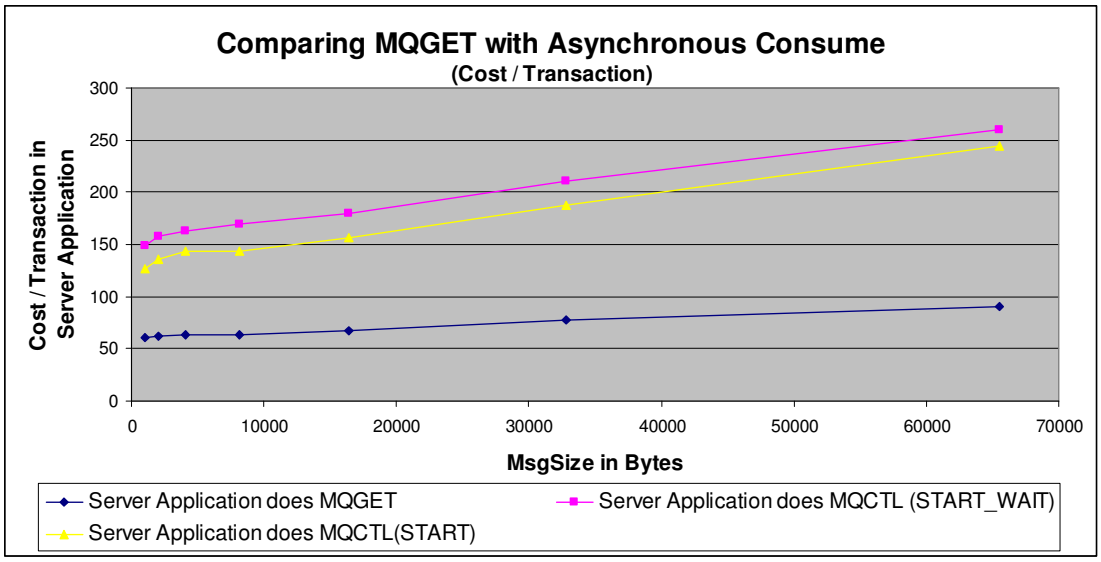
The benefits of asynchronous consume when used by the channel initiator with server-connection channels running with the SHARECNV attribute set to greater than zero can be seen in the charts entitled "Shared Conversations with Read-ahead". Even when DEFREADA is set to off, the rate at which the client is able to get messages far exceeds the rate at which at which a version 6.0 client was able.

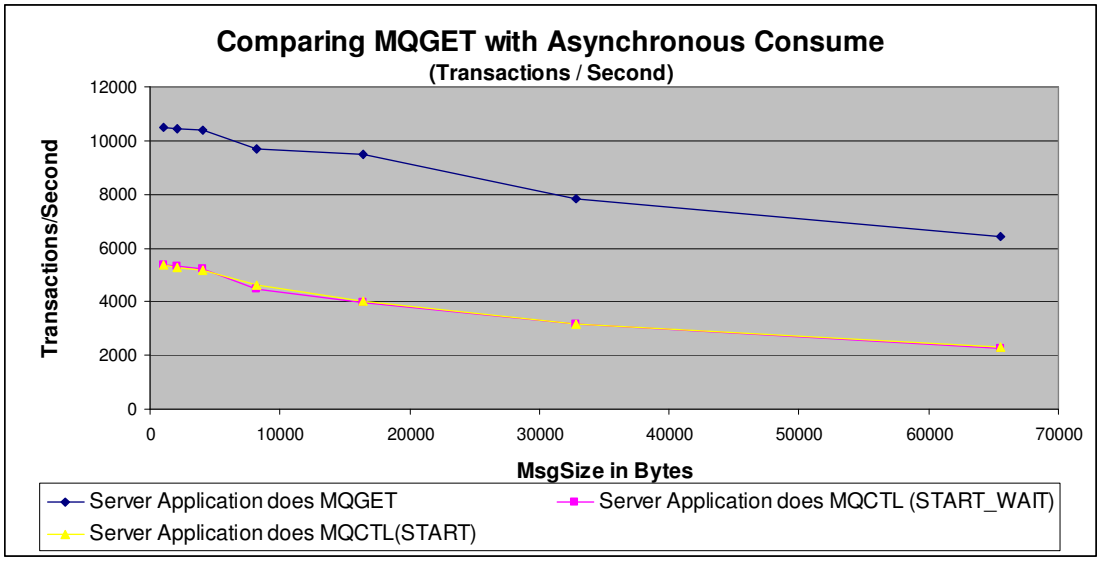
The benefits of asynchronous consume are not so obvious in a batch only environment. The following scenario compares the cost and throughput capabilities of asynchronous consume against MQGET.

- A single batch application is run that puts a non-persistent out-of-syncpoint message to a queue and waits for a reply on a separate queue and then repeats a number of times.
- The message size is varied from 1KB to 64KB.
- A batch server application is run to get the message from the queue and MQPUT1 a reply message to the specified reply queue.

The batch server application runs in 3 modes:

1. Application goes into an MQGET with wait and upon receipt of message, uses MQPUT1 to put a reply message.
2. Application issues:
   - an MQCB "register" to register a message handler routine,
   - MQCTL "start with wait"
   - (when the main thread gets control back, it issues an MQCTL "resume")
   - The message handler routine is driven upon receipt of a message. The message handler uses MQPUT1 to put the reply message and then issues a MQCTL "suspend" to give control back to the main thread.
3. Application issues:
   - an MQCB "register" to register a message handler routine,
   - MQCTL "start"
   - Main thread goes into a sleep for a fixed period.
   - The message handler routine is driven upon receipt of a message. The message handler uses MQPUT1 to put the reply message.





In this environment, asynchronous consume is shown to reduce transaction rates and increase cost per transaction. However the server application does not have any business logic included and in certain circumstances it may be appropriate and beneficial to use the MQCB and MQCTL verbs to schedule a message handler to get a message whilst the main thread continues with business processing.

Using Asynchronous Consume to run a message handler on multiple queues
When an application needs to monitor a set of queues, asynchronous consume provides a simple mechanism to allow the application to get messages from those queues as they are put to the queues.

Consider an application that uses MQGET to monitor multiple queues. The application needs to open all of those queues and then poll each queue in turn using MQGET to see if there are any messages available.

Compare this to an application that uses asynchronous consume to monitor those same queues. The application will open all of the queues, issue an MQCB “register”

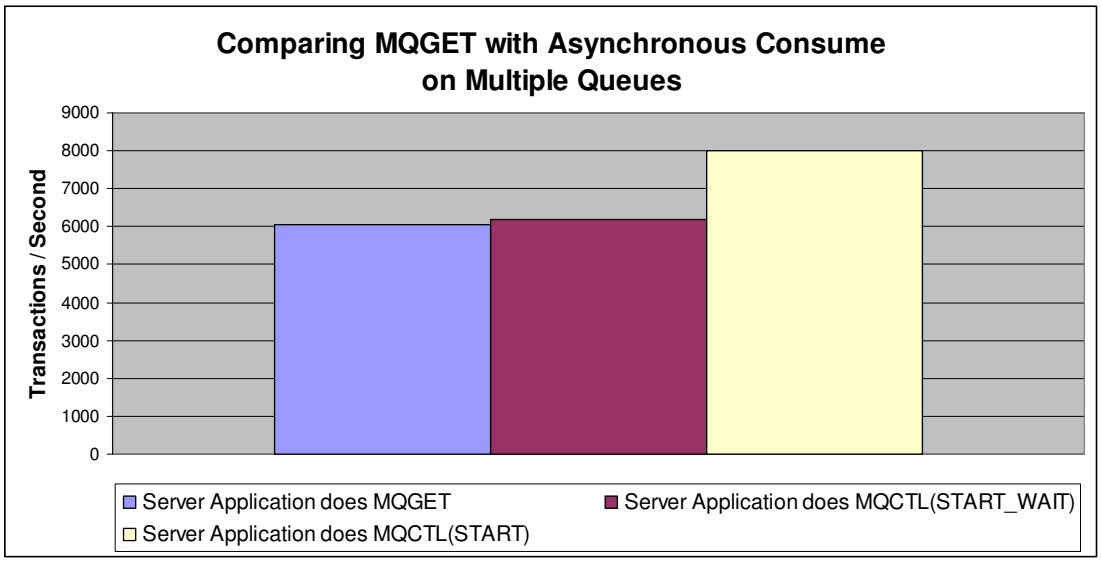
for each of the open handles, issue an MQCTL “start” or “start wait” and the message handler will only be driven when a message is put to any of those queues.

In the asynchronous consume scenario, the message handler is only be driven upon a message being available. By contrast, the application using MQGET may or may not get a message for a particular queue. If two of the queues it is monitoring are getting messages, there are other queues that are being polled unsuccessfully for messages at cost to the application.

The following scenario compares the cost of monitoring multiple queues:

- 10 batch applications are run to put 2KB non-persistent message to separate queues and wait for a reply before repeating the put/get.
- 1 server application is run to monitor 50 queues. The server application will either run using:
    - MQGET with no wait and when a successful get takes place, a corresponding MQPUT1 will be used to put a reply message.
    - MQCB and MQCTL “start wait”, where the message handler issues an MQPUT1 to put the reply message followed by a MQCTL “suspend”. The main thread will restart the message handler by issuing a MQCTL “resume” after each message is processed.
    - MQCB and MQCTL “start”, where the message handler issues an MQPUT1 to put the reply message. The main thread sleeps for a fixed period of time.

.



By using asynchronous consume to monitor a set of queues, the transaction rate has increased by 33% over the scenario that uses MQGET to poll for messages. Using MQGET, the more queues that need to be polled, the slower the transaction rate will become.

In the above case, the cost of polling 50 queues where only 10 are successful is 200 microseconds per processed message, whereas the asynchronous consume case sees the cost of a processed message at 122 microseconds.

CPU cost calculations on other zSeries systems

CPU costs can be translated from a measured system to the target system on a different z/Series machine by using Large Systems Performance Reference (LSPR) tables. These are available at: http://www.ibm.com/servers/eserver/zseries/lspr/zSerieszOS.html

This example shows how to estimate the CPU cost for a zSeries 2064-1C5 where the measurement results are for a 2084-304:

1. The LSPR gives the **2064-1C5** an Internal Throughput Ratio (ITR) of 2.45
(this is for a "Mixed Workload", which we found best fits WMQ in our environment).

2. As the 1C5 is a 5-way processor, the single engine ITR is
2.45 / 5 = **0.49**

3. The "Mixed Workload" ITR of the **2084-304** used for the measurement is **3.60**.
The 304 is a 4-way processor. Its single engine ITR is
3.60 / 4 = **0.90**

4. The **2064-1C5** / **2084-304 single engine ratio** is
0.49 / 0.90 = 0.54 approx

this means that a single engine of a 2084-304 is nearly twice as powerful as that of a 2064-1C5.

5. Take a CPU cost of interest from this report, say **x** CPU microseconds (2084-304) per message, then the equivalent on a 2064-1C5 will be
x / 0.54 CPU microseconds/message

6. To calculate CPU busy, calculate using the number of processors multiplied either by 1000 (milliseconds) or 1000000 (microseconds) to find the available CPU time per elapsed second.

I.E. a 2064-1C5 has 5 processors so has 5,000 milliseconds CPU time available for every elapsed second.

So, for a CPU cost of interest from the report of 640 milliseconds on a 2064-1C5, the CPU busy would be:

640 / (5*1000) * 100 (to calculate as a percentage) = 12.8 %

Measurement Environment and Methodology

Hardware and Software

The hardware configuration was:

- **CPU:** 3-CPU logical partition (LPAR) of a zSeries 990 (2084-332). CPUs were defined as floating but there were always 3 physical CPUs available. Its capacity is similar to that of a 2084-303.
- **DASD:** FICON-connected Enterprise Storage Server (ESS) Model F20.

Software levels were:

- z/OS 1.9
- WebSphere MQ v6 GA
- WebSphere MQ v7 pre-GA levels
- CICS CTS 3.1
- DB2 v8
- IMS v9
- Java 1.5

Client testing was performed on:

- 64-bit RHEL Linux on 4-way XEON 3.66 Ghz processor.