ORACLE
FUSION MIDDLEWARE

An Oracle White Paper
September 2013

Message Sequencing using Oracle Mediator Resequencer

ORACLE

Introduction 1
Basics 3
Resequencer Concepts and Use Cases 5
FIFO Resequencer 5
Standard Resequencer Use Cases 13
Best Effort Resequencer Use Cases 16
Resequencer Anti-Patterns 18
Resequencer Error Handling and Monitoring 20
Performance Tuning of Resequencers 25
Tuning resequencer threads 26
Tuning resequencer datastore 29
HA Considerations 30
Failover 31
Load Balancing 35
A Note on adapter threads in clusters: 36
More Resequencer Use Cases 37
Comparison of Resequencer with Weblogic JMS UOO and UOW 42
UOO vs. FIFO Resequencer 42
UOW vs. Standard Resequencer 43
Summary 44

Introduction

Message Sequencing is often a requirement in Enterprise Application Integration where asynchronous and parallel processing of messages is involved. Even when messages from a source application are **delivered** to the integration layer in the desired sequence, the integration layer could **process** and route these messages to the target applications in an unexpected sequence. This often leads to undesired results such as out of sequence updates and data integrity errors. Parallel processing and asynchronous delivery are the main reasons for this order processing in the integration tier. This is made worse by variation in message characteristics (such as size), variation in processing steps depending on the message content, variation in processing power/availability of resources in different nodes of a clustered deployment, etc

Common workarounds employed to force sequential behavior include singleton adapters in clusters, single threaded components, singleton BPEL implementations, etc. However, these approaches enforce sequential processing of every message delivered to the integration layer severely impacting performance and defeating the purpose of a distributed integration layer.

To address this challenge, Oracle Fusion Middleware SOA Suite provides **Oracle Mediator Resequencer** which guarantees to maintain/restore the desired message sequence in a reliable and robust manner. Oracle Mediator Resequencer (referenced simply as Resequencer in the remainder of this paper) provides both performance and sequential behavior by allowing parallel processing of unrelated entities and enforcing sequential processing of related messages. Resequencing is an option that can be enabled and configured for any asynchronous Oracle **Mediator** service.

Each Resequencer can be configured to best address different functional and performance needs. This paper will provide common use cases for using a Resequencer, best practices when using a Resequencer, configurations that allow tuning the Resequencer, considerations when handling error scenarios, HA and failover, etc. This paper will also shed some light on

the internal workings of the Resequencer which will be helpful in configuring and tuning the Resequencer to perform efficiently.

In addition to guaranteeing ordered processing of messages that are already delivered in sequence, a Resequencer can also be configured to first build a sequence in cases where messages are delivered out of sequence to the integration layer.

Note: Oracle also provides the Weblogic JMS Unit of Order (UOO)/Unit of Work (UOW) features which provide message ordering capabilities. This feature is highly relevant when using JMS-based integrations. This paper will compare UOO with the Mediator Resequencer at a later point and provide the different scenarios where on is preferred over the other.

This document is written based on Oracle SOA Suite 11gR1 version 11.1.1.7. All information in this paper is intended for informational purposes only.

Basics

At the outset the Resequencer performs single threaded processing. However it limits the single threaded processing only to a subset of messages that are intended to be processed in a sequence. For example if a CRM application sends a stream of customer updates, then multiple updates for the same customer are processed sequentially, while updates for other customers happen in parallel. Each subset is called a 'Group' in a Resequencer. Since different groups are processed in parallel, the overall throughput is significantly high compared to a fully single-threaded implementation .

Resequencer enforces asynchronous message processing and is a database-centric implementation. Incoming messages are stored, grouped, sequenced and processed leveraging the database. Resequencer has its own threads that pick up these sequenced messages and invoke downstream services asynchronously. At any point in time, only one thread processes messages of a certain group. The number of such threads determines the number of groups that can be processed in parallel. Figure 1 illustrates these concepts at a very high level where A, B and C are three different groups and A1, A2, B1, B2, C1 and C2 are input messages. In this case the Resequencer is configured to have two processing threads.

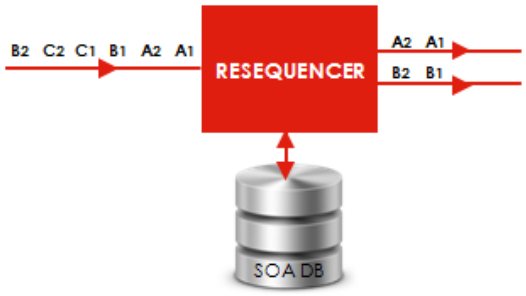

Figure 1: Resequencer Groups

A Group is identified through a Group ID. In the case of a customer update message, the Group ID will be the Customer ID. When configuring a Resequencer, the Group ID has to be identified as an element from the input XML payload. This is shown in figure 2 where it can be seen that the Customer ID is selected as the Group ID in the mediator service definition page. During runtime, when messages arrive at the Resequencer, the Resequencer creates a Resequencer Group in the database for each distinct customer ID. All subsequent messages for that customer are then processed through this Group ID.

With this fundamental mechanism established let us look at how a resequencer builds a sequence.

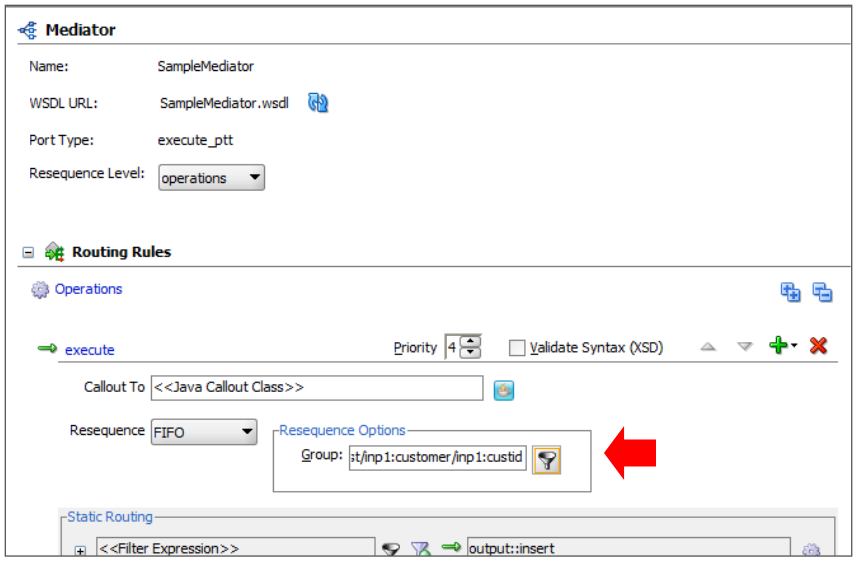

Figure 2: FIFO Mediator Definition at Design time using Jdeveloper

A Resequencer can operate in different modes which will determine the logic used for resequencing. There are three distinct modes that can be used.

- **FIFO** – In a FIFO Resequencer, the resequencing is based on the arrival time of the messages into the Resequencer. If two messages for the same group arrive at the Resequencer it is guaranteed that the message that arrived first is processed before the second message.
- **Standard** – In a Standard Resequencer, the resequencing within a group is not based on the time of arrival but instead depends on a sequence ID identified in the input payload. In this mode, the messages for a given group can arrive at the Resequencer in any order, but the Resequencer guarantees that the messages for that group will be processed 'strictly' based on the sequence ID. Like the Group ID, the Sequence ID is also identified as an element in the input XML message during Resequencer configuration.
- **Best Effort** – In a Best Effort Resequencer, the resequencing is based on a Sequence ID similar to the Standard Resequencer. However, while the Standard Resequencer processes strictly in the order of contiguous sequence IDs, the Best Effort Resequencer will process available messages in increasing order of sequence IDs (not necessarily contiguous) at pre-determined intervals.

The next section of this document will discuss some common use cases of using these three modes of Resequencers. More information about Resequencer modes is available in the official documentation at http://docs.oracle.com/cd/E28280_01/dev.1111/e10224/med_resequencer.htm#CHDBEBGE

Note: Resequencing can be enabled or disabled for every mediator service individually during design time. In fact resequencing can be enabled and disabled selectively for every operation of a mediator service. For each service or for each operation, a different Resequencer mode can also be chosen. This

configuration can be specified at design time only and cannot be modified at runtime. The Mediator Resequencer configuration on Jdeveloper is shown in figure 3.

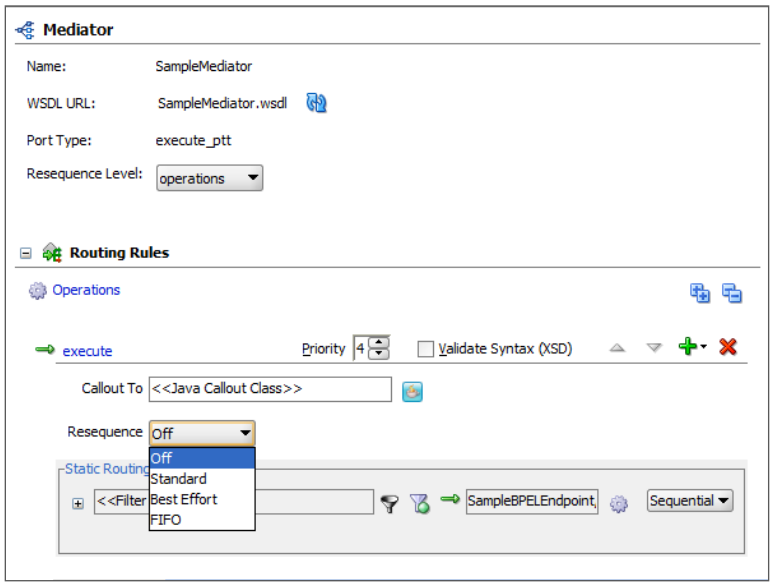

Figure 3: Resequencer modes and Resequencing level

Resequencer Concepts and Use Cases

This section will provide use cases for each type of Resequencer. In discussing these use cases, additional Resequencer concepts and configurations will be explained.

FIFO Resequencer

A common use case for resequencing is when there is a need to avoid an earlier message overwriting the later message.

Use Case – Order Capture FIFO Resequencer Scenario

Assume an Order Capture scenario, where orders created or updated in the CRM System should be delivered to the Order Management System via an Integration layer. Assume that the integration involves several processing steps. When high volumes of orders are involved, delivering the order updates in the correct sequence becomes a challenge. This could lead to updating the Order system with outdated information or referential integrity error. Figure 4 illustrates a sample implementation.

In this scenario, when invoking BPEL, messages are initially submitted to the BPEL internal queue from where the BPEL invoke threads process these messages asynchronously through multiple BPEL instances. This multi-threaded behavior is the default behavior for asynchronous BPEL interactions to provide guaranteed delivery. Since there are multiple BPEL instances working in parallel, out of

sequencer processing is inevitable. As illustrated in the figure, the 4 BPEL threads will process 4 of the 6 messages in parallel. However since create messages can take more time to process than the update messages due to variation in message size or due to additional BPEL logic, update messages can be processed ahead of create messages through other threads available. In the below illustration, while create message for customer A is getting processed, another thread processed the update message for the same customer. This will lead to referential integrity errors.

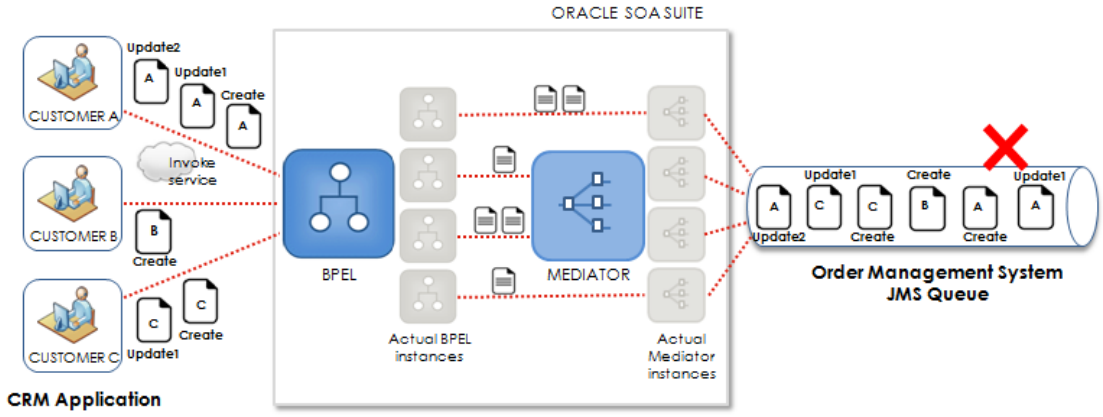

Figure 4: Order Capture Scenario – Out of Order processing

A FIFO Resequencer can handle such a requirement since logically update messages are generated only after create messages. A mediator Resequencer can be added prior to the BPEL component as shown in figure 5. As soon as the Resequencer receives the messages, it sorts the messages into multiple groups based on the Group ID. Since this Resequencer is configured as FIFO it will ensure that messages for a specific Group are processed sequentially in the same sequence that it was initiated. Multiple Groups equal to the number of customers are created and processed in parallel. This is shown in figure 5.

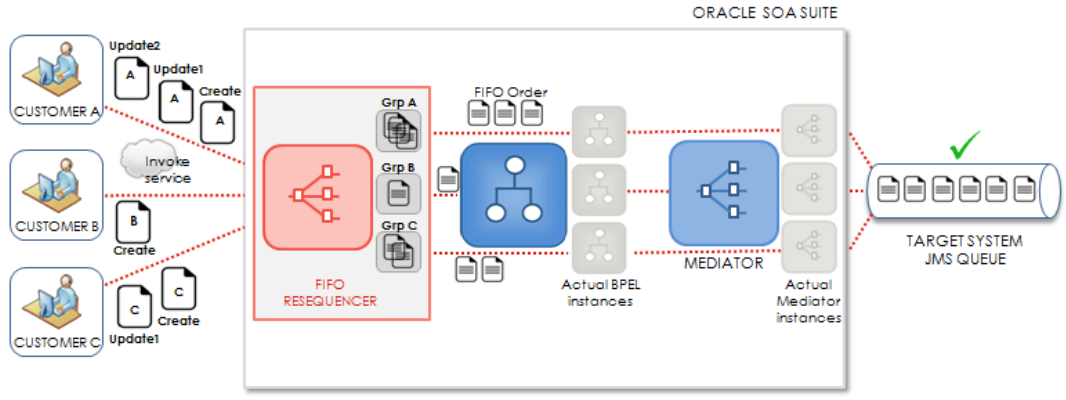

Figure 5: Sequenced Processing of Messages

In this case, there are three groups each with a group ID corresponding to the Customer ID. Messages for each Group A, B and C are processed in the same order. Remember that the final sequence of messages in the JMS queue will preserve the sequences within A, B and C but not across A, B and C i.e., both [A1, A2, B1, C1, A3, C2] and [A1, A2, C1, C2, B1, A3] are valid outputs.

From figure 5 it may appear that if there were 'n' groups, all these 'n' groups would be processed at the same time. However that is not true. The number of groups processed at a given time depends on the number of Resequencer 'worker threads' configured. Figure 6 illustrates the **Phases of a Resequencer execution** which explains this behavior

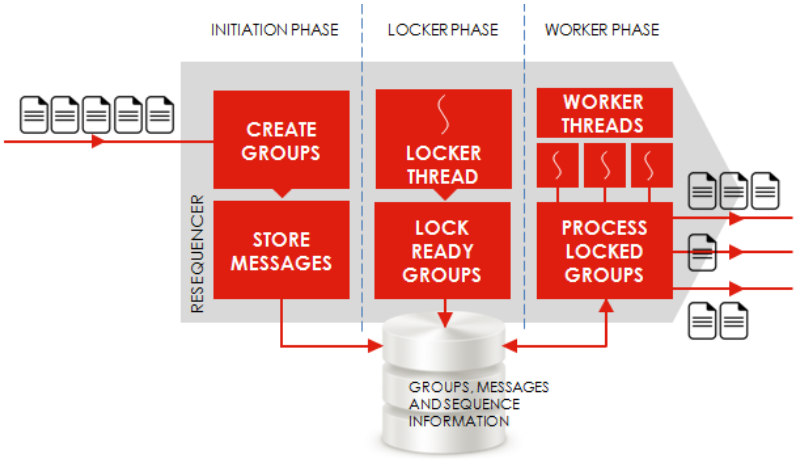

Figure 6: Phases of Resequencer execution

There are three distinct asynchronous phases within a Resequencer when processing messages

- **Initiation Phase** – In this phase the incoming messages are received by Resequencer and the Group ID is extracted from these messages. Unique Group IDs are stored in the Resequencer tables and the corresponding messages are also stored. When subsequent messages for the same group arrive, these messages are stored with reference to the existing Group IDs. The creation time for these messages in the Resequencer table is used as the basis for FIFO resequencing. In case of Standard and Best Effort Resequencer there is additional logic in the tables that builds a valid sequence based on the sequence ID field. Essentially all the sequence related information is built and maintained at the Resequencer table level.
- **Locker Phase** – In this phase a single Resequencer locker thread locks multiple Groups in the database that are 'ready' to be processed. The readiness of a group depends on the Resequencer mode. In case of a FIFO Resequencer whenever a group contains at least one new message it is deemed ready. In case of a Standard Resequencer the availability of the message next in sequence signals the readiness. In case of a Best Effort Resequencer there could be an optional user configured 'time window' after which the group is deemed ready. The locker thread runs continuously and in each transaction it locks all ready groups at a given point in time and signals the worker threads to act on the locked groups
- **Worker Phase** – In this phase the groups that are already locked are processed by the Resequencer worker threads. As part of Resequencer configuration, users can define the

number of these worker threads. Each worker thread will pick a particular locked group and process all messages for that group one after the other based on the sequence related information stored in the Resequencer tables. At the end of processing all available sequenced messages for the locked group, the worker thread unlocks the group allowing for new messages for that group to be resequenced again. After unlocking the group, the worker thread becomes free to process another available locked group.

Based on the description above it is clear that the number of downstream invocations is controlled and limited by the number of worker threads. For example, even if there are several locked groups, if the number of worker threads is 3, then only 3 groups are processed by the Resequencer in parallel.

The worker threads are configured in the enterprise manager FMW control page at the mediator engine level. Figure 7 illustrates the same.

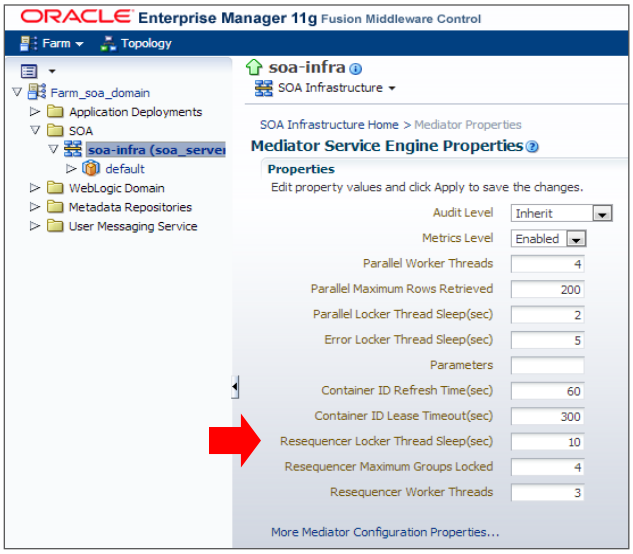

Figure 7: Resequencer Worker threads configuration

Additional Resequencer related configuration seen in this figure and tuning guidelines will be discussed in later sections.

Note: As shown in figure 3 earlier, resequencing can be enabled or disabled for each mediator service operation level however the mediator settings in figure 7 are the mediator engine level. Therefore if there are three different mediator services, each having an operation and enabled for resequencing then the 3 worker threads are shared across all the three Resequencer instances. It is important to understand this when tuning the Resequencer which will be discussed later.

Importance of Transaction boundaries

Understanding of transaction boundaries is very critical to any SOA design. Transaction boundaries often define persistence points and error recovery points. In the context of Resequencers it is particularly important since these are the boundaries within which

Resequencers can guarantee a sequence. A Resequencer cannot guarantee a sequence beyond a given transaction boundary.
To understand this better, refer to the Resequencer block diagram in figure 6. Each of the three phases in the figure runs in a different transaction. Focus on the initiation phase and the worker phase.

In the **Initiation phase**, the source invokes the mediator with a message and this message and the corresponding group information is stored in the Resequencer database tables. This entire activity happens in a single transaction. Each such message is inserted in its own transaction. The arrival time of a message is the time at which this transaction inserts the input message into the Resequencer database. This arrival time is the basis for message ordering by a FIFO Resequencer. The Resequencer however cannot guarantee that the messages arrive into its database tables from the source system in the actual desired order. For example if there are some network delays, or if there are additional components prior to the Resequencer which cannot sustain the desired order or if there is multi-threaded behavior, then a FIFO Resequencer will only be able to maintain the order only as received by it. This design consideration is highlighted in another use case later in this section.

In the **worker phase**, the worker threads pick up the sequenced messages and process until the transaction is committed say when encountering another asynchronous point such as a Queue. Once the transaction is committed, the Resequencer boundary is reached and the Resequencer is not responsible for the sustenance of the sequence from there on. For example, if the boundary is a queue and there are parallel threads that pick up messages from the queue there on, then there is a good chance that the messages will become out of order once again at a later point. This will be illustrated in the upcoming use cases as well.

Consider another order processing use case similar to the one discussed earlier in figure 4. As a slight variation, instead of invoking a BPEL process, assume that the CRM application posts several Sales Order messages to a source JMS queue and an inbound JMS adapter picks up these messages and hands them over to a BPEL process which eventually reaches a target Order processing system. This is shown in figure 8.

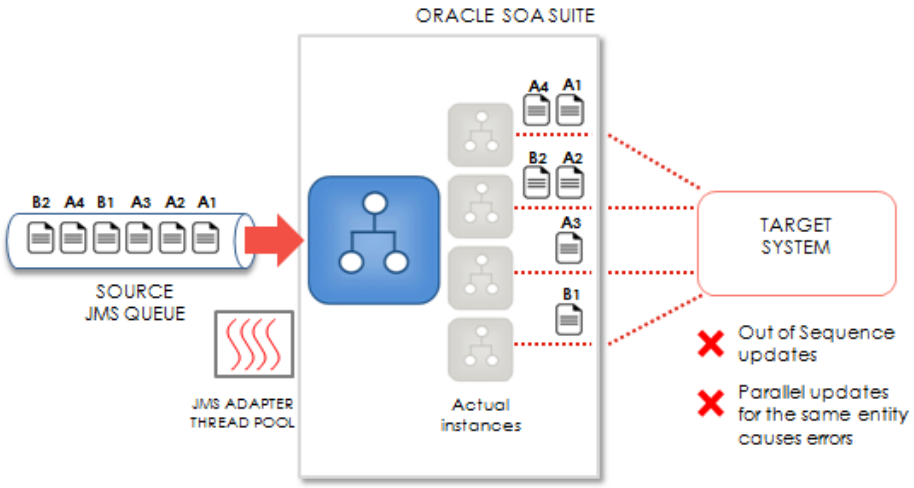

Figure 8: Out of sequence Sales Order Processing without Resequencer

Clearly there is scope for out of order processing of Order updates as already discussed earlier in this paper and and hence a FIFO Resequencer will be used (prior to the BPEL process)

The new flow is shown in Figure 9. The JMS adapter posts messages to the Resequencer which is able to group and sequence the messages. In this case there are two customers therefore 2 groups being processed. However as seen in the figure, the BPEL process still processes these messages in 3 different BPEL instances. This leads to messages reaching the target out of order and also causes messages for the same entity to be processed in parallel.

The main reason this is happening is because of the transaction boundaries on the worker phase. In this case, every BPEL process that has an asynchronous interface, stores the incoming message in a delivery queue before it uses its own BPEL invoke threads to process these messages asynchronously. From a Resequencer worker thread point of view, its transaction (and therefore the scope of sequencing) is complete when it successfully posts messages into this internal BPEL delivery queue. After this point, the BPEL invoke threads process these messages in parallel based on the number of BPEL threads that are available.

In fact this would've been **the same situation with the example in figure 5** earlier but this detail was suppressed at that time for the sake of simplicity.

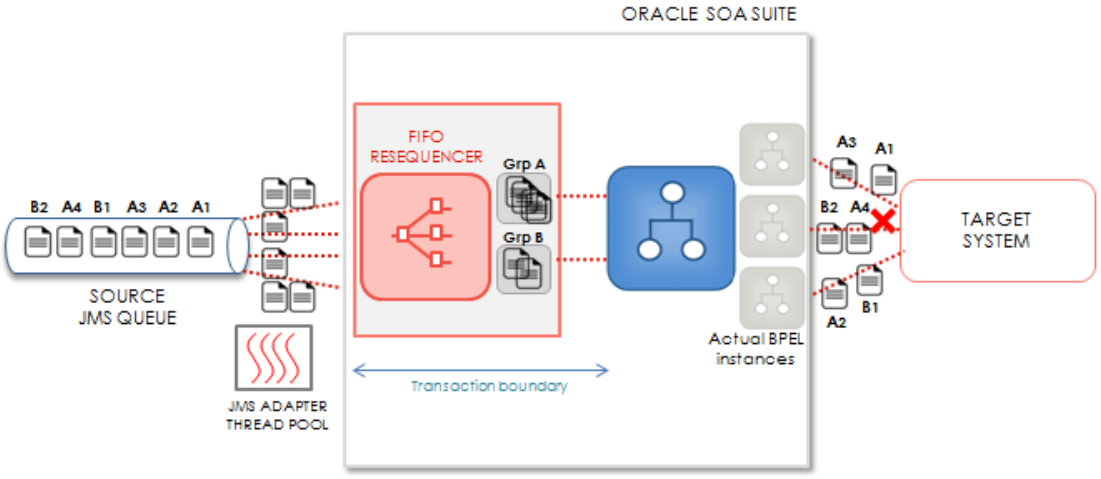

Figure 9: Out of sequence Sales Order Processing with Resequencer

A simple way to avoid this situation would be to ensure that the BPEL processing activity happens in the same worker thread instead of using a new BPEL thread. BPEL configuration provides the oneWayDeliveryPolicy property which can be configured to 'sync' at an individual service level such that this internal BPEL queue will be bypassed and the BPEL process will be called in the same worker thread.

More information about this property is available at
http://docs.oracle.com/cd/E21764_01/core.1111/e10108/bpel.htm#autoId8

Note: Instead of an asynchronous BPEL process, a persistence point can be reached by the worker thread in various other scenarios. Some examples are Mediators with parallel routing rules, Queues and Topics, etc. In general asynchronous processing designs should be adjusted to minimize the overall persistence points along a single flow. If additional persistence points are necessary before reaching the target then additional Resequencers can be employed across these additional transaction boundaries.

Figure 10 illustrates the flow with the BPEL process configured to use the same worker thread to process the messages through the BPEL process until reaching the target. The new transaction boundary is indicated. Interestingly, the FIFO Resequencer still cannot guarantee the order of the messages as shown in the figure.

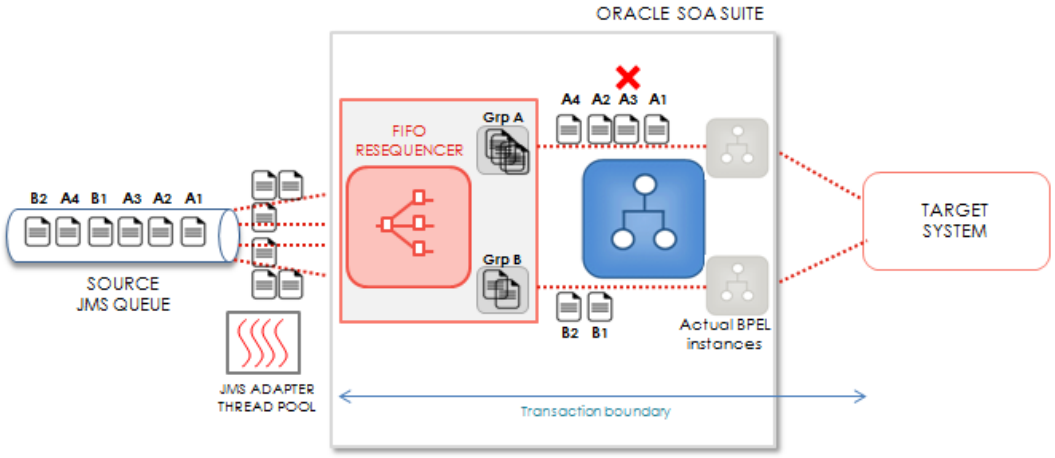

Figure 10: Out of sequence Sales Order Processing with Resequencer and Transaction Boundary

Although messages are grouped and processed one after the other using a single worker thread, the sequence of the messages itself is determined by the arrival time of the messages into the Resequencer during the Initiation phase. This was discussed in the section on transactions earlier in this document. In figure 10, although messages are sequenced in the JMS Queue, since there are multiple JMS adapter threads which poll messages and invoke the FIFO resequencer, there is no guarantee that messages will be created in the FIFO resequencer in the same order in which it was stored in the JMS Queue. In this case the resequencer received messages in the incorrect order (A1, A3, A2, A4) in the initiation phase itself and simply maintained the same sequence. To overcome this, JMS adapter will be made single threaded so that messages are posted to the resequencer in the same order in which it is stored in the JMS Queue. This is shown in figure 11.

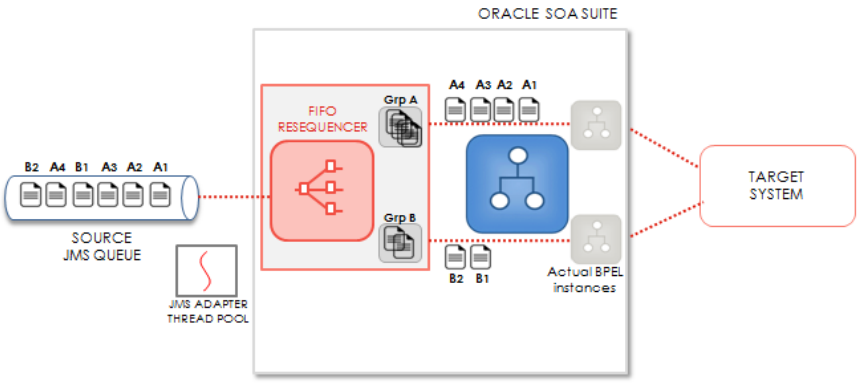

Figure 11: Fully Resequenced Sales Order processing

It may appear that introducing a single threaded adapter will impact the overall performance. While a slight performance downgrade is expected, it should be noted that the downstream worker thread processing is still multi-threaded. In addition to that it must be noted that the worker threads will work

longer in each transaction when compared to the JMS adapter which simply consumes the message and posts it to the resequencer table. As long as the upstream single threaded adapter post messages fast enough to keep the downstream resequencer threads busy, introducing single threaded adapters should not contribute to a serious bottleneck. Performance considerations such as these will be discussed in the Performance section and HA section later in this paper.

> **A Note on Errors**
>
> In any Resequencer mode, if a worker thread fails to process a message, then the entire Group will be marked as error-ed and no further messages will be processed for that group until a manual action is taken by an administrator. This is important to ensure that the messages are processed in the same sequence. Until an administrator takes an action, the Group will not be locked or worked upon. However, the initiation phase will continue to happen and so new messages for that group are never lost. Error scenarios, Monitoring and Recovery methods are described in detail in the Error Handling section later in this document.

Standard Resequencer Use Cases

In the order processing use case discussed above it is assumed that the source JMS Queue already contains the messages in the desired order. In reality, messages may be produced into this source JMS Queue by a CRM application which performs its own processing prior to producing these update messages in the JMS Queue. There could be variations in this processing time due to resource related delays, different workflows, errors, presence of multiple instances of the CRM application, etc., causing the CRM application to produce integration messages into the JMS Queue in an incorrect sequence. Since a FIFO Resequencer can only guarantee ordering based on the arrival time into the Resequencer, it cannot help in restoring the sequence in this scenario.

Use Case – Order Capture Standard Resequencer Scenario

For this modified scenario, a Standard Resequencer can be used. A Standard Resequencer does not sequence based on arrival time but instead depends on a Sequence ID field identified in the input payload. Sequence ID should be a number. For an Order payload, this could be the Order version number while the Group ID is the Order ID. In this example, it is assumed that the Order version number is assigned using a sequence number within the CRM application and therefore indicates the desired sequence of processing. Figure 12 illustrates the scenario. In the figure assume that A and B are the Group IDs and the numbers that follow them are the sequence IDs.

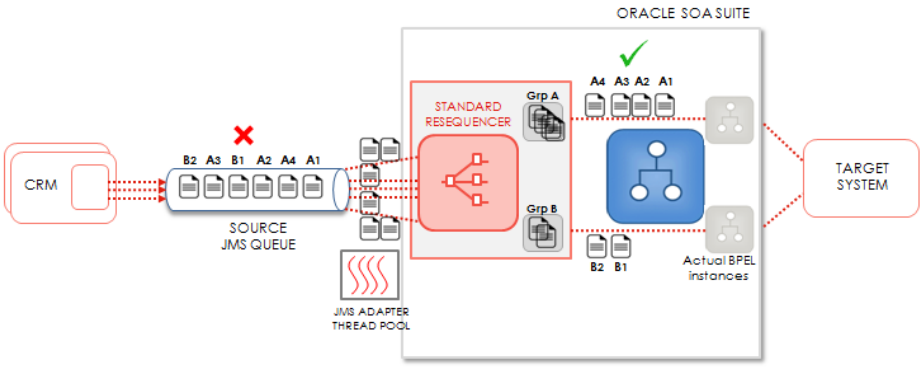

Figure 12: Sequencing using Standard Resequencer

The Standard Resequencer will be configured as follows during design time.

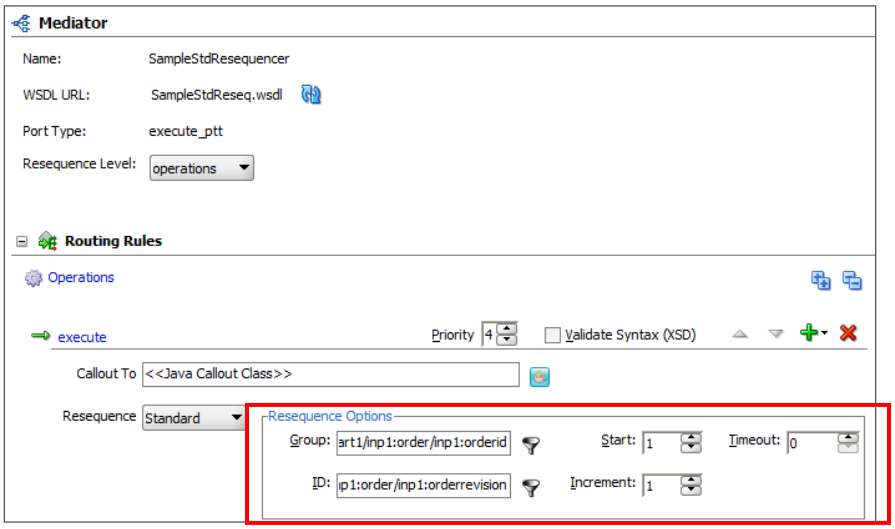

Figure 13: Standard Mediator Definition at Design time using Jdeveloper

Apart from the Group ID and the Sequence ID, there are other options that need to be configured to let the Resequencer know how to interpret the sequence ID. Since the sequence ID is a number, the Resequencer must understand how the sequence gets incremented. This is indicated through the 'Increment' property. A value of '1' indicates that the two sequential messages differ by a value of 1. For example messages with order version numbers 2, 3 and 4 are considered sequential with increment 1. A 'Start' value indicates the first version number to be processed. So even though order version 2, 3 and 4 form a sequence, the group will not be processed until order version 1 is received.

Even after the 'Start' message is received, the Resequencer will halt processing if it misses any sequence id. For example if 1,2,3,4 and 6 arrive, the Resequencer will not process 6 until it receives 5.

It is important to remember that once the start value is available, the group is deemed ready. From then on the locker will lock the group as long as it finds a sequence of valid messages. For example assume that at time't' the locker found messages 1 and 2 available. This signals the group is ready and it

will be locked. Once the worker phase processes these messages, the group is unlocked and the locker again checks for the unprocessed messages in that group. Assume that this happens at time 't+1'. Assume there are no new messages available for the group. This group is therefore not locked. Other groups that may have new messages may get locked in this cycle. Again at t+2, the locker looks into this group. By this time, 3, 4 and 6 may have arrived. The group will be locked. But the worker will work only on messages 3 and 4 on this group. Message with version number 6 won't be processed until 5 is received.

If message with sequence ID 5 doesn't arrive, the group will keep waiting for this message until a timeout value defined in the 'timeout' setting. If this timeout is set to 100 seconds, then if message 5 is not received until this time, then the group will be error-ed and manual intervention will be required. This manual intervention can include the skipping message '5' from the sequence, if the administrator is convinced that version 5 is not going to arrive and it is acceptable for version 6 to be processed. Such cases are covered in the Resequencer Error scenarios and Monitoring section later in this document.

The default value of the 'timeout' is zero, which indicates that the timeout is disabled .i.e. the Resequencer will indefinitely wait for the missing sequence number to arrive. In the case of Order scenario discussed above, it is possible that some version numbers can arrive much later because it could require some manual approval as part of a workflow. In such cases the timeout may not be set or can be set to a high value.

Maintaining a strict sequence and having lower timeout may be very important in some other use cases. For example, assume a banking system, where many transactions which are initiated during the day are all processed at the end of that business day by integrating with another funds transfer system. A single banking account could have initiated multiple transactions including both credit and debit. Assume account ID is the group ID and the transaction ID is the sequence ID. Although all transactions for all accounts will be processed in bulk in the same hour, it is very important to process these transactions strictly in order. Hence a Standard Resequencer is highly desired. In this case, since all transactions are processed together, there is no reason to set a high 'timeout' for sequences to arrive. In fact, administrators would like to quickly react to missing sequences so that the rest of the transactions are not blocked.

Like a FIFO Resequencer the Standard Resequencer will also halt the processing of the group if there is any error in processing the messages by the worker threads. Irrespective of newly available sequences, the Group will continue to be blocked if the previous messages threw an error. An administrator has to take a manual action in this case. This will be covered in the Resequencer Error Handling and Monitoring section later in this document.

In figure 12, another key aspect to note is that the adapter threads do not have to be limited to 1. In FIFO since the logic depended on the arrival time, it was important to have the adapter threads set to 1. In the case of Standard Resequencer, since the sequence ID solely determines the order, this restriction is unnecessary.

Best Effort Resequencer Use Cases

A Best Effort Resequencer uses a Sequence ID similar to a Standard Resequencer except that it does not enforce a strict contiguous sequence like a Standard Resequencer. The Best Effort Resequencer does not wait for a 'Start' sequence ID nor does it wait for a next message based on a sequence 'Increment' value. In fact both these settings are not applicable for a Best Effort Resequencer since it simply orders available messages (based on a sequence ID) at pre-determined intervals.

Use Case – Order Capture Best Effort Resequencer Scenario

Assume the scenario discussed in the Standard Resequencer scenario in figure 12. As a slight variation assume that order version numbers cannot be guaranteed to be contiguous. This may be because some order updates are not published to the JMS Queue by the CRM system. This may also be because some complex order types may have a different number versioning mechanism.

In this case, a 'Start' or 'increment' value cannot be assumed. If say, an order version number 2 was discarded and never submitted to the JMS Queue, then the Resequencer will keep waiting for order 2 until time out occurs and manual intervention takes place instructing it to processes subsequent version numbers. This is one scenario where the Best Effort Resequencer is suited since it does not depend on a fixed start and increment value. Instead, the Resequencer simply groups messages belonging to A and B and processes them in sequence at pre-defined intervals. This is shown in fig 14

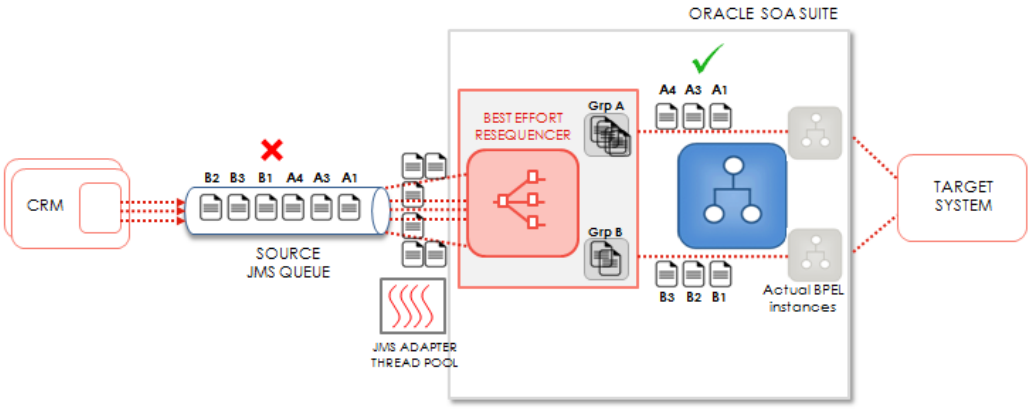

Figure 14: Sequencing using Best Effort Resequencer

The Resequencer is designed in Jdeveloper as shown in figure 15

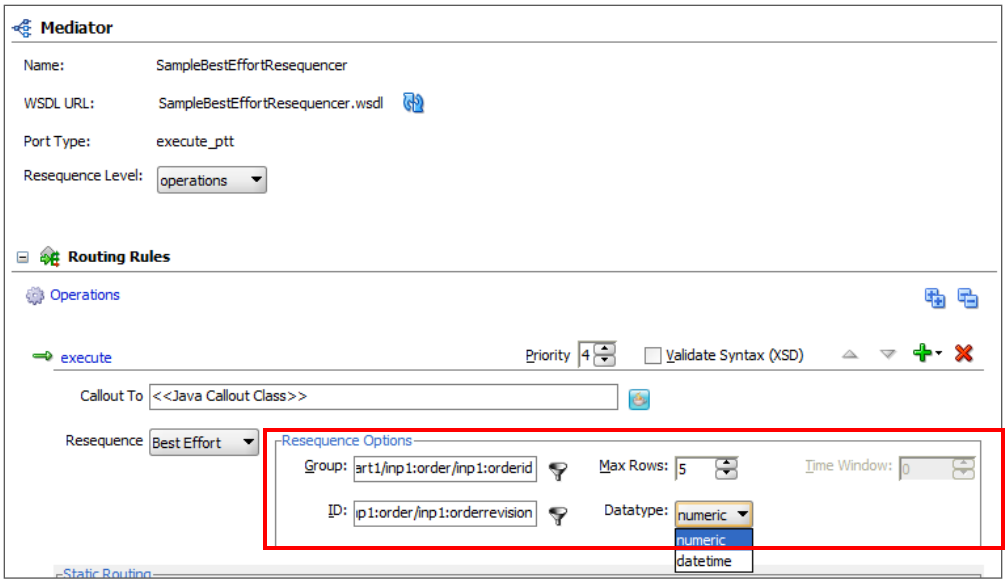

Figure 15: Best Effort Mediator Definition at Design time using Jdeveloper

Notice that apart Group ID and Sequence ID, there is a Datatype field as well. This field specifies the type of data in the Sequence ID. Since order version number is numeric, this configuration is selected as numeric for this example. The Best Effort Resequencer also allows for using a datetime field in the payload as a Sequence ID. For example an Order update time field in the order payload can be used a sequence ID. This allows the best effort Resequencer to work process messages for a particular group ID in an increasing order of timestamps of each message completely independent of the version number.

However since the Best Effort Resequencer does not wait for a start message, more caution is required. Assume that the version number is used as the Sequence ID. As discussed earlier in the context of Standard Resequencer, the create order message with version 1 may take a longer time, say 50 secs, in the custom workflow of the CRM before the message reaches the JMS Queue. This leads to a situation where order version 2, 3 and 4 (which take only 5 secs in the custom workflow) will be available sooner for processing for the Resequencer. Since the Best effort Resequencer doesn't use a start and increment setting, it will simply attempt to process 2, 3 and 4. This is definitely not desirable.

To overcome this, the Best effort Resequencer configuration provides a 'time window' configuration, which can be used to define a wait time before which the locker locks the group. In this example, it will be prudent to set the 'time window' to 50 secs so that the Resequencer allows enough time for the version 1 to also be posted. This way, 1, 2, 3 and 4 can be processed together. Note that this locking behavior is different from other Resequencers since in this case even though a valid sequence is formed the group is not locked until the time window expires.

Even when time window is set to 50 secs, sometimes the version number 1 can take slightly more than 50 secs due to variation in processing time, network delays etc. For this reason, in addition to the 'time window', an additional 'buffer window' property can be set at the mediator engine level. If the buffer

window is set to 10%, then the locker waits for additional 10% secs (5 secs in this case) before locking. In these 5 seconds no new high sequence ID values are accepted. Only values lower than the existing highest values are accepted for processing. For example, if 2, 3 and 4 arrive within 50 secs and version 1 arrives in the 53rd second, then version 1 will be included for processing. However if version 5 arrives in the 51st second, then it will not be considered for processing. Notice that if version 1 arrives in the 57th second, then 2, 3 and 4 will be still be processed at the end of the 55th second, leading to data integrity errors. This is why 'time window' only allows controlling the resequencing behavior to a large extent but cannot guarantee a sequenced output.

In addition to the 'time window' configuration there is another configuration that allows controlling the resequencing behavior. This is the 'max rows' property shown in figure 15. Note that, the 'max rows' property or the 'time window' property can be set, but both cannot be set together. Max rows property achieves a similar goal of waiting for more messages. This property allows controlling the number of sequenced messages to be processed in one worker phase. Referring to the block diagram in figure 6, the 'worker' phase only processes 'max rows' number of sequenced messages in one locking cycle even though more sequenced messages are available for the locked group. The main goal of this limiting is to allow for a 'good' sequence to arrive within the time the current group is being worked upon. So if messages 1, 2, 3, 5, 7, 8, 10, 11, 12, 13 are available, and if max rows is set to 10, then all records will be processed by the worker thread in one cycle. In the x seconds it took to process these records, the missing sequence IDs 4, 6, and 9 could have arrived. In the next locking cycle the worker threads will process 4, 6, and 9 resulting in version 9 overwriting version 13.

However if the max rows is set to 3, then in the time the worker thread processes 1, 2 and 3, there is a good chance that 4 and 6 could arrive from the source system. That way, the next locking cycle could lock 4, 5, 6 correctly. This delaying also postpones updating the highest sequence ID to the very last. Again, like the 'time window' approach this approach is also simply a best effort and cannot guarantee an exact order. The max rows property works well when large numbers of messages are processed in a short time window and when these messages are already in near sequence.

'Max rows' or 'time window' should be tuned as desired to achieve the best results. It must be remembered that irrespective of the option chosen, there is an inherent delay in processing messages. The official documentation at http://docs.oracle.com/cd/E17904_01/integration.1111/e10224/med_resequencer.htm#autoId15 illustrates the usage of time window and max rows with more samples.

Like with other Resequencers, processing halts during faults. The errored groups have to be manually resolved as discussed in the Resequencer Error Handling and Monitoring section later in this paper.

Resequencer Anti-Patterns

Although Resequencers provide several desirable benefits as discussed above, it should be remembered that when Resequencers are introduced in designs it creates additional processing logic, requires additional storage, requires additional monitoring, administrative actions and involves additional configuration and tuning.

Therefore, one must be careful about evaluating the need for Resequencers. In some scenarios although asynchronous updates take place and sequential processing may be desired, a Resequencer can be unnecessary or undesirable.

Use Case – Too few updates

Assume an 'online order capture system' where customers simply choose a simple product, fill out order details and click on submit. Although this order may be modified later by the customer, such updates may be rare since the device is a simple known product. Even if updates were to occur they may not occur in quick succession. In such a case, a Resequencer may not be necessary since the primary purpose of Resequencers is to only sequence updates which occur close enough to cause an overlap. It makes little sense to introduce Resequencers when there is only one update every day or when updates are known to happen once every few hours.

There may be some corner case scenarios. For example, if updates were to happen in quick succession (a customer changed mind immediately or clicked update twice), out of order processing may occur. Similarly fresh order update can be issued when an earlier order update has faulted. Although a Resequencer is designed to address these two scenarios, it may be prudent to handle these rare exceptions manually rather than investing on a Resequencer.

Use Case – Queue vs. Resequencer

Assume a scenario where employees of an organization place orders for certain office supplies with a vendor. Assume that multiple such organizations are placing orders with this vendor and all this information is stored in the vendor's CRM system (in a JMS Queue). At the end of every business day, the vendor processes these orders. Refer to figure 16. Assume that the vendor maintains one account for each organization (shown as A, B in the figure) and that the vendor's order system cannot handle multiple orders (shown as numerals next to A and B) in parallel for these accounts. In addition the order numbers for each customer are not expected to be in sequence since order numbers are not generated per customer. Based on our earlier examples, a Best Effort Resequencer seems to fit this scenario.

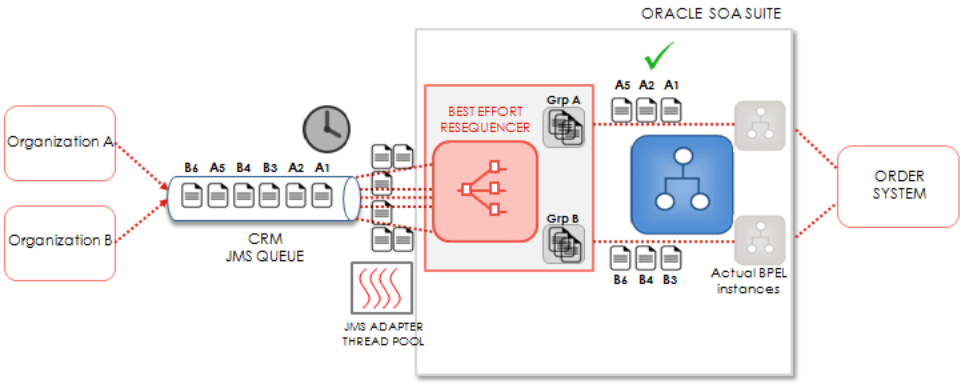

Figure 16

However there are some major disadvantages when using a Resequencer here. For the Resequencer to prevent multiple Sales Orders being processed for the same customer the Group ID should be the Customer ID and the sequence ID should be the order ID. Now, even if one order for a particular customer fails, then all orders for that customer are blocked. In this case it is not acceptable because

- The orders themselves are independent of each other. Although there is a need to maintain single threaded processing, there is no reason to block one order due to a fault in another order.
- The order processing happens every night. Hence it adds unnecessary overhead for someone to manually work on the blocked groups to allow other orders to be processed.

As an alternative, then a single threaded JMS consumer can be employed instead of a Resequencer. This will satisfy the requirement of processing only one order at a time for a given customer. In case of faults or exceptions, this consumer will continue to process subsequent orders.

If the numbers of such corporate customers are few, then other mechanisms such as JMS filtering or mediator parallel routing rules can also be employed in addition to single threaded processing. This paper will not discuss these alternatives as they are not entirely relevant.

> **A reminder on the Objectives of a Resequencer**
>
> In general it is must be remembered that a Resequencer achieves
>
> 1. Sequenced processing of related messages, so that an entity on the target is updated by only one thread
> 2. Sequencing of messages in a desired order so that the same entity is not updated with an outdated message.
> 3. Suspending further processing if one message fails

If one or more of the above is not necessary, then Resequencer usage should be reconsidered.

Resequencer Error Handling and Monitoring

An important aspect of a Resequencer is the manner in which error scenarios are handled. In the previous sections it was discussed that if an error occurs when processing a message in a group, then that error will cause subsequent processing of that group to be suspended. The processing will not continue until an administrator acts on this group to perform a remediation action. Similarly if a Standard Resequencer reaches timeout when waiting for a message then the group is timed out. Figure 17 below explains different Group states. The figure is self-explanatory. A Group continuously processes messages until it requires a manual administrator action. This section will describe how these errors and timeouts are handled.

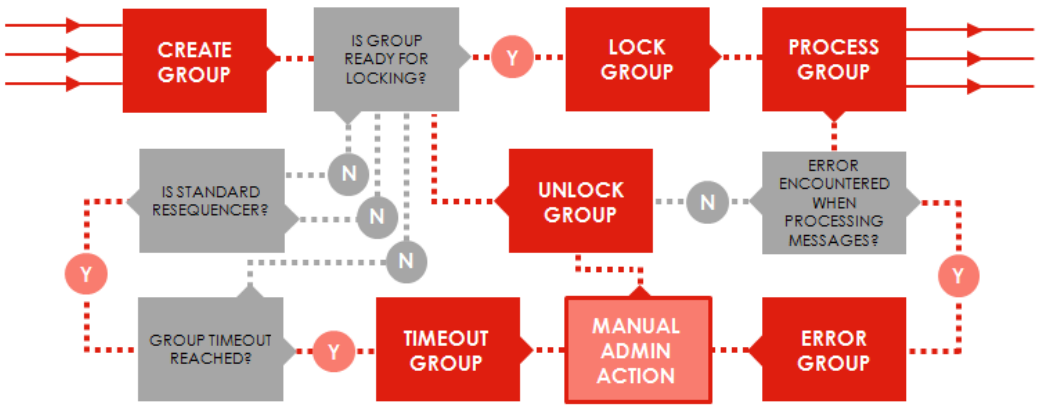

Figure 17: Group States

Before moving on to Error Handling this paper will discuss 'when' Resequencer errors occur. Understanding transaction boundaries is important to determine this. Transaction boundaries were discussed earlier in this paper. Figure 18 below is an extension of the Resequencer block diagram shown in Figure 6. In this figure the transactions boundaries are shown clearly, one corresponding to each phase of the resequencing process.

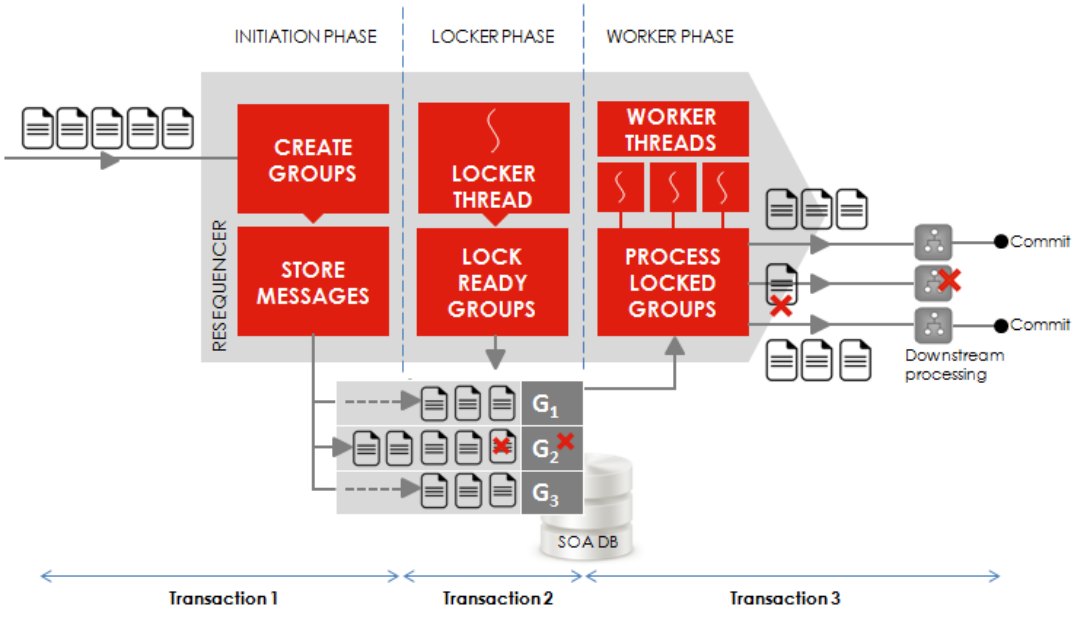

Figure 18: Resequencer Transactions

Focus on transaction 1 and transaction 3 since transaction 2 is internal to the workings of the Resequencer.

Transaction 1 spans from the invoker up to the point where records are committed to the Resequencer tables in the SOA Infra Schema. If there is any error in this process, the message is rolled

back to the caller (say a JMS Queue if messages were posted to the Resequencer from a JMS Queue). These errors should therefore be retried from the JMS Queue level.

Transaction 3 spans from the worker thread picking up messages for locked groups, up to the point where a next commit happens. As discussed in the transactions section earlier in the document, these commits can occur in many scenarios. There are many Transaction 3 one corresponding to each worker thread. If there is any error in processing a message, then that transaction is rolled back and has to be retried from the **Resequencer level**. Retrying/recovering/skipping errors will be discussed soon.

Once an error occurs when processing a message in transaction 3, the entire Group is marked as 'Error-ed' and the locker phase will no longer lock this group. This is shown in figure 18 above, where G2 is marked as Error-ed. It is important to notice that the initiation phase will continue to happen through transaction 1 even though the Group is marked as Error-ed. This way, new messages for that group received from the invoker are never lost. Of course this will build up a backlog as shown in the figure.

The questions that follow are

- How are these errors detected by the administrator?
- How are these errors remediated?

Like for any regular mediator, Resequencer Mediator instances are also shown on the enterprise manager in the instances tab and fault tab of each mediator. Additionally, for a Resequencer mediator, the instance/fault page also displays the group ID for each mediator instance. An administrator can search instances/faults based on a group ID. For the earlier examples, when searching for a particular Order ID, one can simply provide the Order ID and check the status of the instance

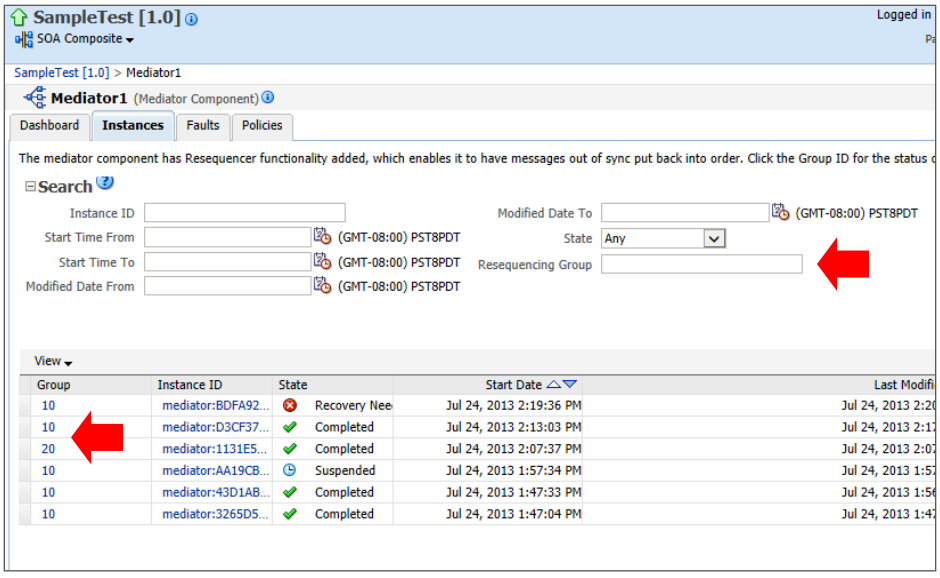

Figure 19: Viewing Resequencer Mediator Instances

Clicking on the group ID provides more information about the group such as status, blocking message, recovery steps etc.

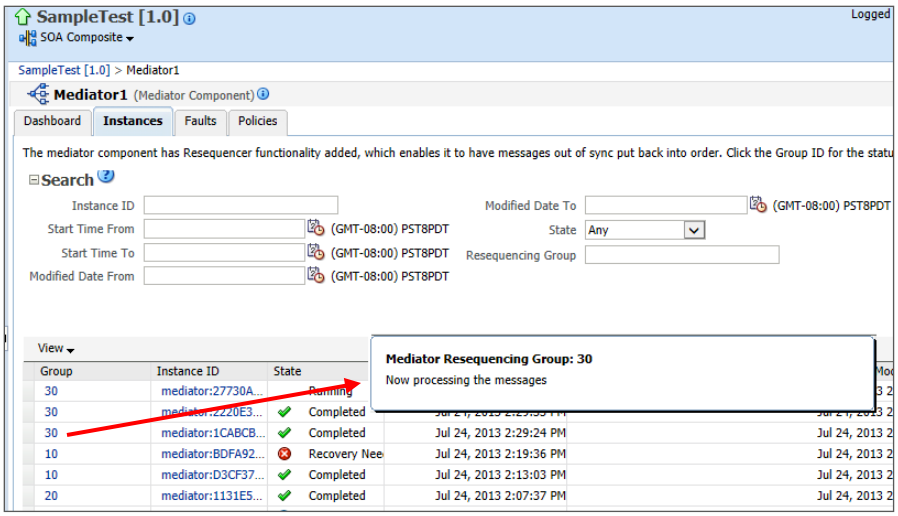

Figure 20: Viewing Resequencer Group Status

Clicking on a suspended group allows for additional actions based on the reason for the suspension. A group can be suspended for the following reasons

- When a message in the group is faulted - This may be because of a business error downstream such as product associated with the Order is not found or any other non-system error. When a group is faulted, the administrator can recover the instance as shown in figure 21. Recovery may include actions such as Replay, Abort, etc. On completion of the recovery activity, the group will automatically be marked again as open, so that the locking thread can lock this group again for processing. Since new messages were always stored for this group, once the group is locked, these backlog messages will continue to be processed in the desired order.

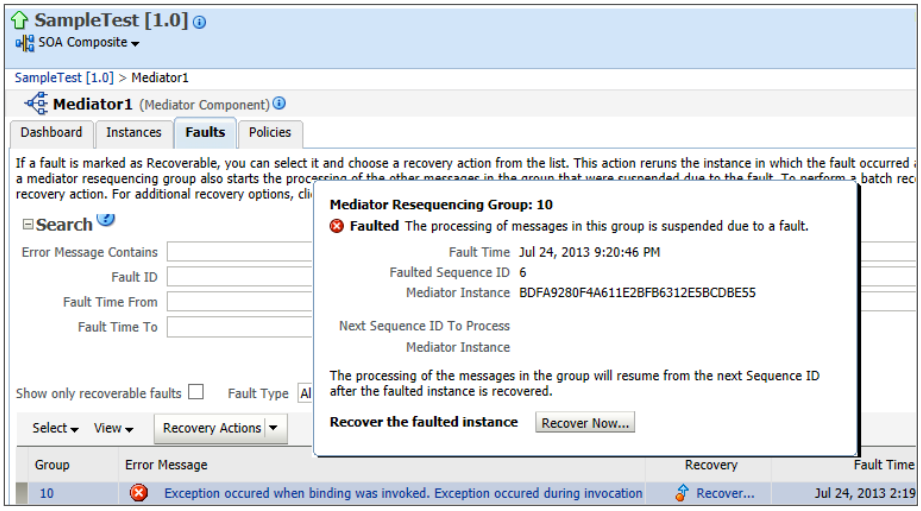

Figure 21: Recovering Faulted Resequencer Instances

- When a message in the group is faulted due to a system error - This can be because the target system is unavailable. A pop-up allows you to simply retry the transaction. In this case, the retry will succeed when the target system becomes available. Once the retry is successful, the Resequencer allows continued processing for that group similar to the case of business faults.
- When a group times out waiting for a next message (such a missing order version number) in the case of Standard Resequencer – This case is slightly different from the other scenarios because there are no faults or retries. A pop-up as shown in Figure 22 allows you to skip the missing message. If the missing messages were to arrive later, they will not be processed. Remember that this timeout is defined per mediator service where Resequencer is enabled and not at the engine level.

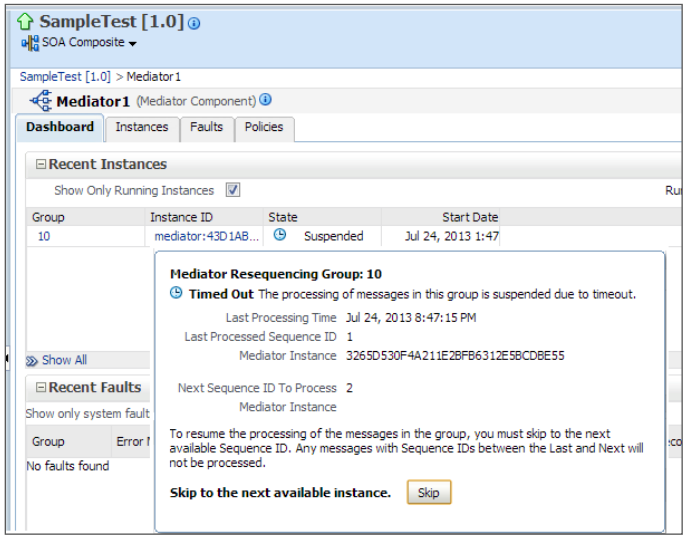

Figure 22: Skipping Message sequence in Standard Resequencer

More detail about error and monitoring is provided in the official document at http://docs.oracle.com/cd/E23943_01/admin.1111/e10226/med_mon.htm#BABIJCEE

Note: Resequencer mediators do not have fault policies like the Parallel routing mediators. However in addition to recovering mediators using the EM console as discussed above, Resequencer APIs maybe used to lock, unlock, skip messages etc. Currently these APIs are not documented but are being used within AIA implementations. The snippet below shows how Resequencer APIs are used within AIA products .There may be future product enhancements and additional documentation in this direction.

```
java.util.Hashtable jndiProps = new java.util.Hashtable();
java.lang.String weblogicUser =
java.lang.String weblogicPassword =
jndiProps.put(javax.naming.Context.SECURITY_PRINCIPAL, weblogicUser);
jndiProps.put(javax.naming.Context.SECURITY_CREDENTIALS,weblogicPasswor
d);
```

```
loc =
oracle.soa.management.facade.LocatorFactory.createLocator(jndiProps);
//get the group first
oracle.soa.management.internal.facade.mediator.MediatorInstanceImpl
instance =            new
oracle.soa.management.internal.facade.mediator.MediatorInstanceImpl();

instance.setComponentType(oracle.soa.management.util.MediatorInstanceFi
lter.COMPONENT_TYPE_OPERATION_SEQUENCING);
instance.setResequencerType("FIFO"); // options are "FIFO",
"BestEffort" and "Standard"
instance.setComponentDNwithoutLabel(serviceComponentDN);
instance.setOperationPerformed("execute");
instance.setGroupId(groupId);
Object[] arr={instance};
oracle.soa.management.facade.mediator.MediatorGroup
group=(oracle.soa.management.facade.mediator.MediatorGroup)
(loc.executeServiceEngineMethod(loc.SE_MEDIATOR,"getGroup",arr));
Object[] arr1={group};
// this is to skip the faulted message
loc.executeServiceEngineMethod(loc.SE_MEDIATOR,"unlockGroup",arr1);
```

Also, when using Oracle AIA Foundation Pack/PIPs, in addition to using the EM console and the APIs, the AIA resubmission utility can also be utilized. AIA Resubmission Utility is documented at http://docs.oracle.com/cd/E23943_01/doc.1111/e17366/chapter17.htm

Performance Tuning of Resequencers

Since Resequencers execute through multiple asynchronous phases (as discussed in the block diagram in Figure 6 and Figure 17) and since it involves additional database operations, there is an added performance cost when using Resequencers. The goal is to keep the Resequencer throughput as high as possible even while it is sequencing and single threading. When designed and tuned well, the performance of Resequencers should be significantly better when compared to a fully single threaded design and should be slightly worse when compared to an asynchronous design which doesn't implement sequencing/single threadedness. With that in mind, look at some performance tuning guidelines.

A good design is the most basic step towards better performance. Some design patterns and anti-patterns were discussed earlier in this paper and there would be some more examples towards the end of this paper.

Resequencer tuning will focus on two broad areas

- Resequencer threads
- Resequencer datastore

Tuning resequencer threads

The Resequencer works in phases as discussed earlier and shown again in Figure 23. Begin by focusing on the worker threads. To a large extent, the number of worker threads defines the overall throughput of the resequencer.

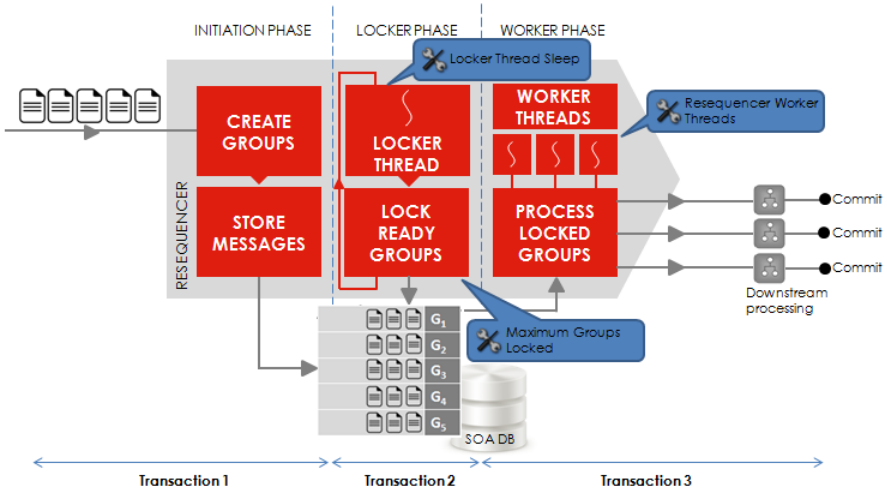

Figure 23: Resequencer Threads and Properties

The locker phase may lock many groups, but the number of worker threads define the actual number of groups that will be processed at a given time. This number is defined by the **Resequencer Worker Threads property**, which is set at the Mediator Engine level as shown in the Figure 24. As discussed earlier, it must be remembered that these Resequencer worker threads are shared by all resequencer enabled mediator instances. These worker threads themselves are obtained from the default work manager for SOA Suite and are dedicated for this purpose.

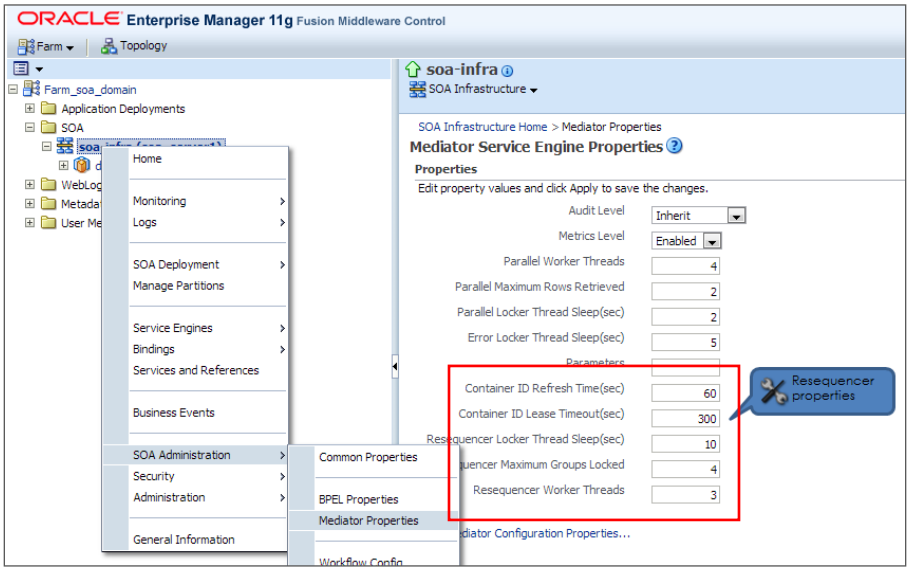

Figure 24: Resequencer tuning properties

For example, in Figure 24 it is shown that the number of worker threads is set to 3. The impact of this setting is shown in Figure 23. Three worker threads lead to only 3 groups being processed at a given time. That means that messages for these three groups are processed sequentially in the desired order. If each of these three groups have 5 messages each then all the 5 messages are processed before the worker threads are released for it to process the other groups that are locked.

All locked groups are available to be processed by the worker threads. However, the number of groups locked itself is configurable. That is not all the ready groups in the database are locked and made available for the worker threads to process.

The locker thread runs continously as long as the server is alive and in each cycle, it locks only a certain number of groups. The number of groups locked in each locker cycle is defined by **Resequencer Maximum Groups** locked property shown in Figure 23 and Figure 24. In this case, this property is set to 4, which implies that in one cycle the locker thread only locks 4 groups and makes them available for the workers to act upon. Once the locker completes this action it starts its next cycle and in this cycle it locks the 5th group and upto 3 more new groups that may have arrived in this interval. When the locker thread cannot find any groups to lock, it goes to sleep for an interval of **Resequencer locker Thread Sleep** property.

It is important to throttle using **Resequencer Maximum Groups** locked because if too many groups are locked, then it could overload the resequencer.

Note: In case of clustered deployment, these numbers apply to ever node in the cluster.

Having described the properties and its impact this section will focus on some guidelines for tuning these. (Note that Contained ID properties shown in Figure 24) are applicable only in the case of Resequencer deployments in a cluster and will be described in the HA section of this paper.

It may appear that tuning increasing the **Resequencer Worker Threads** property will allow more groups to be processed concurrently and therefore increase the throughput. Although true to a certain extent, increasing this number beyond a point will increase the load on the downstream systems (which could be more SOA processes or Target applications). This increase in load can cause resource contention and therefore impact negatively on the overall throughput of the system.

Another important aspect to consider is the time taken for a worker thread to complete a single transaction. As seen in the previous examples, if downstream BPEL processing and target system invocation happen in the same worker thread, then the worker thread is held for that long. Since the worker threads enforce sequential processing, subsequent messages for the same groups are held back. In the example shown in Figure 23, if each of the 3 worker threads have 5 messages each and each message takes 10 secs to process, then the remaining 4 messages of that group are held until that 10 seconds. It takes a total of 50 secs to process that group before the worker thread can work on the next locked group. Some customers incorrectly percieve this as messages being "stuck" in the resequencer. This has nothing to do with the performance of the resequencer and cannot be improved by tuning the resequencer since it is the expected behavior of a Resequencer. In this case it is important to reduce the processing time of the downstream processes. Increasing the number of worker threads will improve the throuput because more groups are processed in parallel.

When increasing the number of worker threads, it must be made sure that there are enough number of groups available for the worker threads. For example if there are too few groups locked, it doesn't matter if the number of worker threads are simply increased. Therefore the **Resequencer Maximum Groups locked** should be increased to provide enough groups to keep the worker threads busy.There is no simple way to decide the number mainly because the worker threads typically work longer than the locker threads.

One good approach to tuning would be to start with **Resequencer Maximum Groups locked = Resequencer Worker Threads**. The actual value could be a number such as 20. This starting value can be 50 or 100 for large loads and very large loads or can be 5 for small loads. With an arbitary number set, one should monitor if sufficient overall throuput is achieved by the flow. Increase the worker threads to increase the throughput. If sufficient througput is not reached try increasing the maximum groups locked to provide more groups for the resequencer to work on. Tuning is an iterative process.

One important factor which cannot be controlled through the resequencer properties is the rate at which incoming messages arrive into the resequencer. If too many messages arrive into the resequencer, then a lot of messages could be "stuck" in the resequencer before it can be locked and processed. The number of messages fed into the resequencer should be controlled, if feasible. For example, when using a JMS adapter or a file adapter prior to the Resequencer, the adapter threads can be used to control this inbound flow. Similar to how Maximum Groups Locked cannot be simply derived from the Worker threads configurations; there is no easy way to derive the number of inbound adapter threads based on the resequencer parameters. For example, if a JMS adapter simply consumes messages from a JMS Queue and posts it to the mediator, then the inbound transaction will be significantly short lived compared to a the worker thread transaction which spans across downstream BPEL processes and Target applications. Hence even if the inbound adapter threads are much lower than the worker threads, they may be working fast enough to keep all the worker threads busy. In general for such cases it is advised to start with 1 JMS adapter thread and increase if necessary. Remember, that as discussed earlier, in most cases FIFO resequencers require this single threaded inbound behavior to be able to guarantee sequenced processing.

When messages are posted into a resequencer directly by a source system (instead of an intermediate JMS Queue, Database Table etc) there is no easy way to throttle the input. In such cases the worker threads and maximum groups locked can be changed to keep up with the the average input rate. High Availability deployments will also assist in increasing the possible throughput. If no alternative is possible, the design can be changed to introduce an intermediate JMS adapter.

Essentially the **inbound adapter threads, maximum Groups locked and the Worker threads** properties must be tuned together to achieve the maximum throughput. At the same time the tuning exercise must also aim at maintaining a consistent flow of messages into and outside the resequencer.

The AIA scenario tuning whitepaper at http://www.oracle.com/us/products/applications/aia-11g-performance-tuning-1915233.pdf shows how tuning Resequencers is one of the tuning activities when tuning an integration flow for higher throughput. In addition to Enterprise Manager SOA Pack, the paper describes additional tools and approaches that can be used to monitor the overall throughput

.The paper also highlights the importance of throttling adapter threads and how increasing the threads in a system beyond a point introduce a negative effect.

The above tuning parameters are common for all types of Resequencers - FIFO, Standard and Best Effort. In addition to that, there are some things to remember specific to a particular Resequencer type. For a Standard Resequencer, it must be remembered that the throughput is also a function of the probability of the next sequence ID arriving soon enough. In case of a Best Effort Resequencer, the messages must arrive as close as possible for optimal results. If the messages do not arrive in quick succession, then set the value of the maxRowsRetrieved parameter to 1 for that Resequencer instance. This value limits the actual number of messages processed for a group to 1 message, although many messages exist for the locked group. Having the value set low as 1 allows more time for the next message in sequence to arrive so that it is available within the next processing cycle.

Another approach that was discussed earlier for Best Effort was to increase the 'time window' parameter to delay the locking of the group and thereby ensuring that a better sequence is formed in that time. However this is not suitable for systems where the response time is important. For example, in the example described earlier in this paper, the 'time window' was set to 50s, which implies that message processing will be delayed by as much as 50s which may be completely unacceptable from a response time perspective. For these systems the time window has to be set to a much lesser value and in addition the incoming flow rate can also be throttled (for example by tuning adapter threads) to increase the probability of obtaining a better sequence. Again, since this is best effort processing, a sequence cannot be guaranteed.

Resequencer tuning is a critical exercise in large deployments. In some of large Oracle deployments, Resequencers resequence up to 1 million messages per hour. In such cases tuning efficiently can produce a significant improvement in overall throughput.

Tuning resequencer datastore

Since Resequencers use the database to persist messages and groups, perform sequencing etc, database performance becomes an important factor in the overall throughput.

Resequencers use two database tables viz., mediator_resequencer_message table, which stores the metadata about the message to be resequenced and mediator_group_status table, which stores the group related information. All Resequencer modes rely on these two tables.

Note: The actual payload is not stored in the mediator_resequencer_message table but is instead stored in the mediator_payload table.

The volume of the mediator_resequencer_message table is directly proportional to the volume of messages processed by a flow. If a flow contains a Resequencer and processes 10,000 messages an hour, the number of records in the mediator_resequencer_message table will increase by 10,000 records per hour. The volume of the mediator_group_status table is directly proportional to the number of distinct group IDs. So if there is an average of 4 messages per group, then the number of groups will be 2500. Although the volume is largely predictable, since the Resequencers perform complex sequencing logic using these tables, it is in the best interest to control the growth of these tables.

The records continue to keep growing until they are purged. Unlike some of the SOA tables, the amount of data that enters the Resequencer tables cannot be controlled by turning instance data capture off, setting log levels; setting in-memory persistence, etc., since all data is relevant for performing the sequencing functions. Oracle provides scripts for purging the Resequencer tables. These must be used in conjunction to the overall Oracle SOA purge strategy defined at http://www.oracle.com/technetwork/middleware/bpm/learnmore/soa11gstrategy-1508335.pdf

However, if a large amount of load is processed, the Resequencer table may need more frequent purging when compared to the other Oracle SOA tables in the SOAINFRA schema. For example, Resequencers can show deterioration in processing throughput when the numbers of messages in the table are over a million.

To overcome the need for purging Resequencer very often, Resequencers allow messages to be deleted as and when they are processed successfully. This is available through patch 16602054 on 11.1.1.5.0. The Group records still have to be purged manually. Note that groups have to be purged carefully in the case of Standard Resequencers since the Group defines the start value and the next expected value. One must be sure that more messages of the group will not arrive before purging these groups. Optionally, after the groups are purged, one can manually modify the start values and next values to continue processing.

In addition to the above, the Resequencer tables have to be constantly monitored for performance through general database monitoring and tuning methodologies. Partitioning is currently not supported for Resequencers.

HA Considerations

Having discussed the Resequencer in detail, the paper will next discuss the functioning of Resequencers in a clustered (Highly Available) environment. Clustered SOA deployments are very common in production environments. Like most SOA Suite components, Resequencers also provide improved throughput in a cluster by processing more messages in parallel through different nodes. Resequencer instances run actively on each nodes of the cluster and all mediator settings such as Resequencer Worker Threads, Maximum Groups locked etc, apply to each node of the cluster.

To visualize the functioning of Resequencers in a cluster, the Order Capture Integration Flow that was discussed in FIFO Use case section (seen in Figure 11) will be expanded so that there are have 3 nodes running within a cluster. This is illustrated in Figure 25 below. It shows the Resequencers configured with 2 Worker threads.

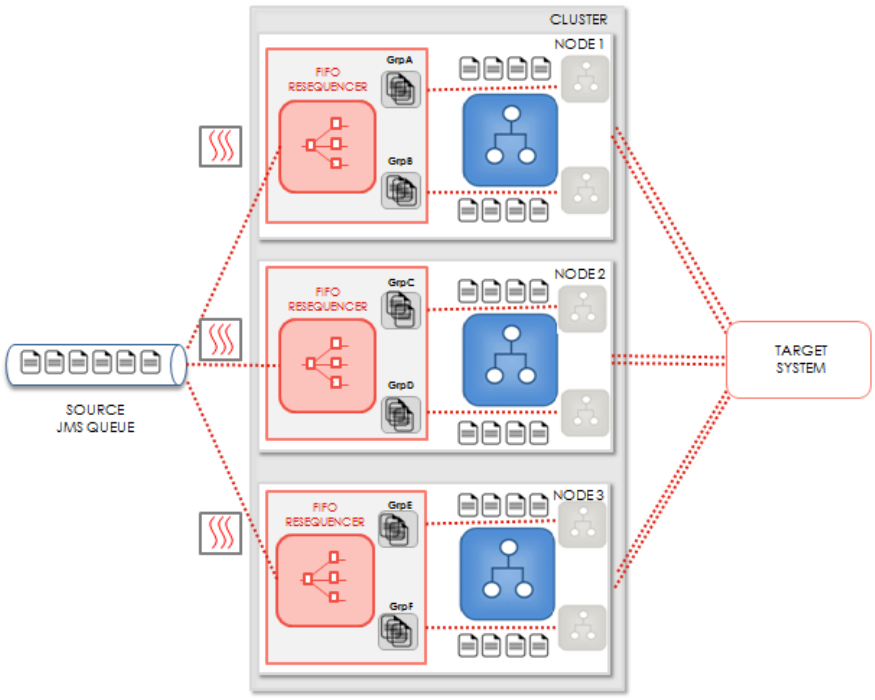

Figure 25: Resequencer in a clustered deployment

In this scenario, assume a constant input rate of Update Order messages arriving into the inbound queue. At any point in time when all the nodes are functioning normally, the Worker threads of all 3 Resequencer nodes are processing Order Updates and routing them to BPEL Component which finally posts them into the target system.

Resequencers are designed to be cluster safe. That is, the sequencing logic is maintained across all the nodes of a cluster. In an 'n' node cluster, even though for each Resequencer definition, there are 'n' Resequencers deployed, if the Resequencer in node 1 is processing a Group, then none of the other n-1 nodes will process messages for the same Group.

This is achieved by pinning a group to particular node so that all messages for that group are always processed through that node. In this manner each node processes an entirely different set of groups at a given time. In the figure 25, Group A belongs to a specific Account (assuming Account ID is configured as the FIFO Resequencer Group ID) and since this group is pinned to node 1, all messages that arrive for Group A are processed using node 1.

Failover

In case of a node failure, groups belonging to that node failover to available nodes so that the processing can continue on these other nodes. The main configurations performed by an administrator in this context of a failover would be the durations of inactivity or refresh which allow the Resequencer to determining that a node (container) is dead. This duration is based on two configurations, namely

Container ID Refresh Time and **Container ID Lease Timeout**. Figure 26 depicts the Container ID Refresh Time for a single node

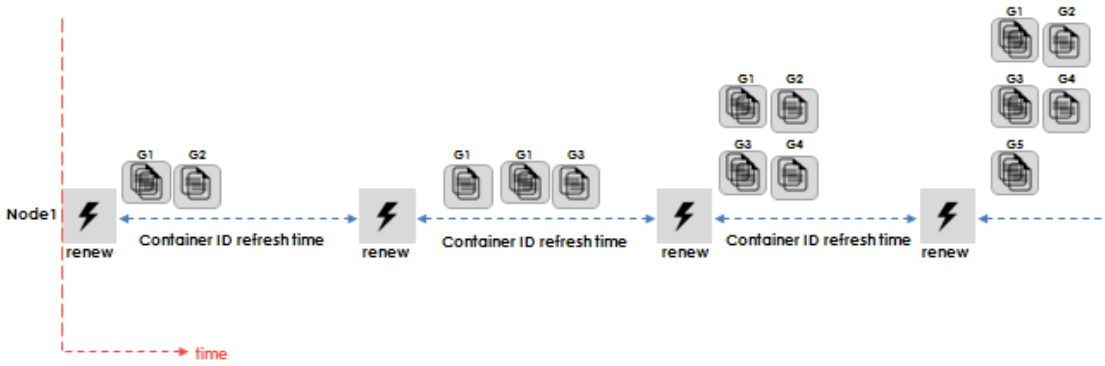

Figure 26: Container ID renewal

Whenever a Resequencer node in a cluster is active, it renews itself with a lease table once every pre-configured Refresh Cycle to signal that it is active. This event is depicted as 'renew' in the Figure 26. Since every active Resequencer Node in a cluster renews itself, the lease table maintains the current view of the Resequencer cluster at any given point in time. The Refresh cycle runs throughout the lifetime of a Resequencer. The Groups are shown to indicate that several groups accumulate over this time.

The time between every renew event is governed by the 'Container ID Refresh Time' setting of Resequencer set in the Enterprise Manager FMW Control Mediator Settings page(Figure 7). The default value of this parameter is 60s.

If a node fails to renew its lease, the node is deemed dead. The most common reason for this is node failure, although other reasons such as database issues, resource starvation, etc., may sometimes cause the node to not renew its lease.

To understand the failover process and the importance of Lease Timeout, multiple Resequencer Nodes should be looked at. This is depicted in Figure 27. It shows 3 Resequencer Nodes of a cluster. In this figure, Node 2 fails after the first refresh cycle.

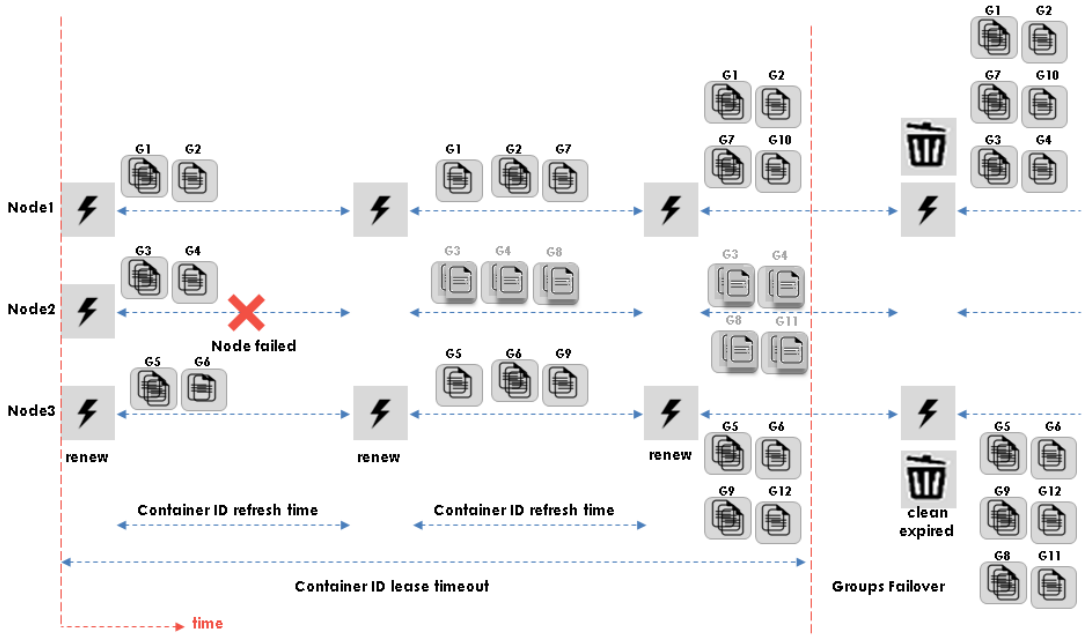

Figure 27: Failover across nodes in a cluster

It should be noted that the cluster may not always immediately detect the failure of a particular Resequencer Node. This is where the **Container ID Lease Timeout** plays a role. This timeout is the amount of time the cluster waits before declaring a non renewing Resequencer Node as expired. In other words, any node missing to renew itself for Container ID Lease Timeout will be deemed expired. Such a node becomes a candidate for failover. In the Figure 27 although Node 2 failed during the first Refresh Cycle, the failover of that node happened at the end of the Lease Timeout period. The Lease Timeout is typically set a multiple of the Refresh Time e.g. Lease Timeout = 5 X (Refresh Time).

This parameter can be configured from the Enterprise Manager FMW Control Mediator Settings page at runtime. Refer Figure 7. The default value of this setting is 300s.

Let us also see the Failover of groups in the context of the Order capture use case in Figure 25. Imagine that one of the nodes in the above setup, say Node 2 fails to renew due to a container crash during processing. Assume that the Worker threads of Node 2 are processing groups and that Locker thread has also locked more groups for processing when the Node 2 crashed. This would leave groups that are in the middle of processing pinned to dead node and in a locked status. This is depicted in the Figure 28 below.

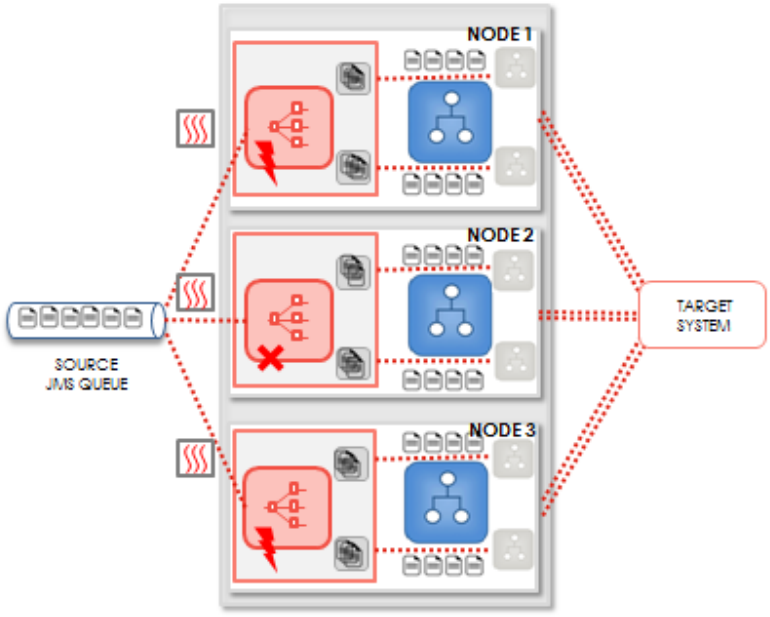

Figure 28: Order capture Resequencer Failover in a cluster deployment

If the Node 2 fails to renew until the Lease Timeout duration, this situation triggers a Node Failover by the other active Resequencers running on Node 1 and Node 3.

During Failover, one of the active nodes will unlock all the groups that are locked by the now defunct Resequencer Node. It will also distribute the ready, error'ed and timed out groups so that they will be further processed by an active node. This is taken care by the load balancing logic of the resequencer. Finally the expired Resequencer Node is deleted from the Lease Table's view of the cluster. The groups that were locked by the defunct container are now unlocked and become eligible to be locked by the Locker Threads of all active Resequencer Nodes. They will then be processed by the Worker threads of the active Resequencer nodes. This is illustrated in the Figure 29. The figure shows the groups GrpC and GrpD which were orphaned by the failed Node2 are owned by Node1 and Node 3 after the failover.

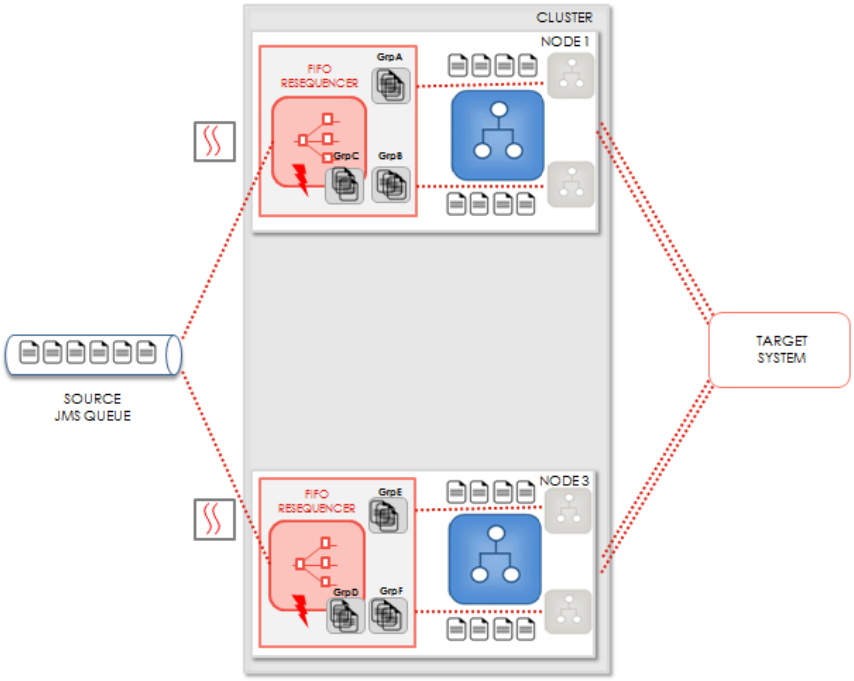

Figure 29: Groups after failover to active nodes

The **Container ID Refresh Time** and **Container ID Lease Timeout** settings should be set with caution. It is obvious that Lease Timeout should be greater than the Refresh Time. A very large value of Lease Timeout (e.g. 20 X (Refresh Time)) however, could delay the failover of groups to active Resequencers. On the other hand, a very small value, say 2 X (Refresh Time) could unnecessarily trigger a failover even if any transient DB error caused the renewal to fail twice consecutively.

Also note that a Failover can cause sudden increase of load on the remaining active Resequencer nodes. This is due to the reassigning of the incoming and existing groups to the active Resequencer Nodes. It is worth noting that the setting of **Container ID Lease Timeout** impacts this failover load. It was noted earlier that Groups continue to be pinned to a failed Resequencer Node even past its failure, owing to the Load balancing of the Resequencer Cluster. With higher values of the Lease Timeout, the node to be failed over accumulates groups over a longer time and hence more pronounced will the increased load on the surviving Resequencer Nodes.

Load Balancing

The Resequencer Nodes in a cluster continuously load balance the number of groups that they lock and process. It should be noted that this Resequencer Load Balancing is provided by the Resequencer Component in addition to the any external load balancing and its primary aim is to balance the load of processing incoming Resequencer Messages among all the active Resequencer Nodes. However at any point of time all the messages belonging to a single group are always processed within only one Resequencer Node, as that is the core idea of sequenced processing of related messages as guaranteed by a Resequencer.

Load balancing can be thought of as a background process that runs on all active Resequencer Nodes. It runs throughout the lifetime of the Resequencer node. It attempts to balance the **new** groups owned by the Resequencer Node before it is locked for processing. Once locked the groups continue to be owned by the node until a failover. Note that Load Balancing is an internal functionality which is inherently available to all Resequencer deployments in a cluster. There is no manual intervention required from the end user to achieve Load Balancing.

In a Resequencer Node Failover scenario, referred in Figure 27, it was noted that the list of active Resequencer Nodes is maintained at the Lease Table. It was also noted that a failed node is cleaned up from the Lease Table at the end of its Lease Timeout period. During Failover, the internal Load Balancing then ensures that the orphaned groups are evenly distributed among the active Resequencer Nodes for further locking and processing

It should be noted however that this inherent Load Balancing and Failover features could have a negative impact as well for certain corner case scenarios. One such scenario is discussed below.

Consider that due to some maintenance, all the Cluster nodes are shutdown and restarted together. When this happens during load conditions, messages will continue to pile up at the input source. During the startup, even though all nodes are started together, one Node starts first and begins to own the ready groups. Since none of the other nodes are still active, it ends up locking the ready groups

In addition to owning all open groups, there are also a number of groups locked for processing by the other nodes when they were shutdown. Based on the failover discussed, the first node now starts to own all these orphaned groups. This could be a huge number, especially in large clusters. Until the other nodes become active on the cluster, this first node starts up and begins to lock and process all of the new and failover groups available for processing.

This can lead to a severe imbalance and performance hit on this particular node. The Nodes that startup subsequently will be also be underutilized. This issue has been addressed as a patch on 11.1.1.5.0 –patch 16602054.

This solution introduces a new configuration called **distributionDelay**. For the time period set as this delay, the nodes don't perform any failover or clean up for nodes deemed "dead". In reality, this delay gives enough time for the rest of the cluster nodes to start up, so that the first node that starts up doesn't end up owning most of the groups. Note that this is applicable only during server startup. Once the server is started up, the **Container ID Refresh Time** and the **Container ID Lease Timeout** logic will apply. This value should be set equivalent to the time difference between starting up the first node and the last node of the cluster. For example if all nodes are selected together on the Weblogic console and started together, this value can be as low as 30-60 seconds. If the nodes are started one after the other, the time taken could increase.

A Note on adapter threads in clusters:

It was learnt in earlier sections that when using JMS Queue and FIFO Resequencer, a single threaded JMS adapter is required. In the above clustered setup in Figure 25, even when the number of consumer

threads is set to just one, there is an N fold increase in the number of Adapter threads on every node, where N is the size of the cluster. In this case 3x3=9 threads across the cluster. Also observe in Figure 28 that this number of consumer threads on each node has dropped to 2 (total 4) after one of the nodes has failed in the cluster.

This is due to a known issue with JMS adapters because it creates a consumer/subscriber for each member of the distributed destination even though the members are local. This may be fixed in the future releases. Due to this issue, when using FIFO Resequencers with JMS Adapters in a cluster there will be an increase in the number of threads. However as long as the number of threads is set low (say 1), the messages arrive spread out and if the number of cluster nodes are less then, the chances of out of sequence processing is very rare. One of the use cases in the next section will provide an example for the same.

More Resequencer Use Cases

Based on our end-to-end understanding of Resequencer concepts below are some examples which are commonly encountered by Oracle customers.

✍ Use case – AIA Order to Activate PIP

Oracle Application Integration Architecture (AIA) is set of Oracle products that provide a jumpstart to an Enterprise Application Integration initiative. Built on Oracle Fusion Middleware and focused primarily on SOA-based integrations, AIA Foundation Pack provides a rich set of integration tools, canonical objects and reference models that allow organizations to quickly build standards-based integrations between their Enterprise Applications. Oracle AIA also offers Process Integration Packs (PIP) which are pre-built integrations built using AIA Foundation Pack and SOA Suite. There are many PIPs offered by Oracle which integrate different Enterprise Applications such as Siebel, E-Business Suite, etc. These PIPs deliver multiple integration flows using SOA components. Some of these integration flows implement the Oracle Resequencer and adhere to the best practices that were discussed earlier. One such use case will be discussed here, to provide a real life example from a PIP that is being used by many customers worldwide.

The Oracle Communications Order to Activate (O2A) PIP integrates Siebel CRM, Oracle Order and Service Management (OSM) and Oracle Billing and Revenue Management (BRM) applications. Siebel CRM captures orders such as a mobile phone connection order and integrates with BRM and OSM to process and fulfill the order. OSM performs order fulfillment and for each order being fulfilled, OSM sends back several update messages to Siebel, indicating the current status of Order Fulfillment.

This flow enables OSM to enrich the Sales Order in Siebel CRM with updates coming from the downstream provisioning systems. These updates provide the Customer Service Representative (CSR) logging in to Siebel with a view of the real time provisioning status of a Customer's order. These update messages contain the overall Fulfillment status such as Progress, Complete, etc as well as the sub-status (called as OSM milestones) such as Shipped, Provisioned, Installed and so on. In this flow,

around 9 such updates are sent as the order proceeds through all the different stages of fulfillment. Customers can introduce finer statuses and increase the number of updates to more than 9. For the purpose of this example, assume that this number is 9.

It is obviously important that the statuses reach the CRM in the correct order. Otherwise, if the 'Complete' message is processed earlier than a 'Processing' message, it could lead to more severe errors and data Integrity issues. Typically OSM sends all these updates in a matter of seconds and since such Telco customers use PIPs under large load scenarios, there is a high chance for messages to be processed out of order when not using a Resequencer. Figure 30 below shows this 'Update Sales Order' integration flow.

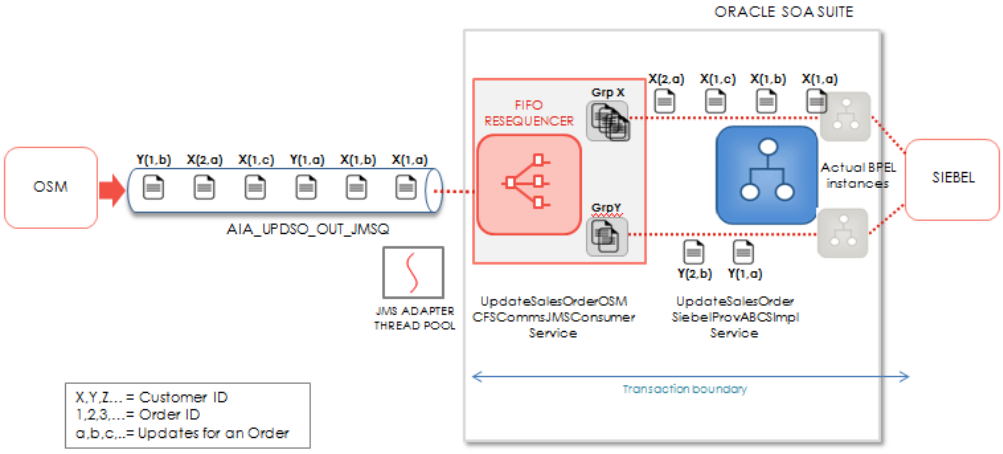

Figure 30: Oracle O2A PIP for communications- Update Sales Order Flow

This is a use case that warrants the use of a FIFO Resequencer. In this case, a FIFO Resequencer guarantees that all Order Updates are processed in the sequence in which they were updated by OSM. Additionally, if a business error or a system error causes an order update to fail in Siebel, then the Resequencer suspends further processing until an administrator acts on the errored group.

Since in this business scenario multiple orders can be modify the same assets on Siebel, the Group ID in this case is selected as the Customer ID (also called the Account ID). This implies that if one order update for a customer fails then all orders updates for that customer are suspended. In the above figure, if X(1,b) message fails then both X(1,c) and X(2,a) are blocked. Here X is a Account ID, (1,2,3..) are the Order IDs and (a,b,c) are updates messages for each order.

Note that there are two transaction boundaries in this scenario, one from the JMS Queue to the Resequencer and the other from the Resequencer worker thread to Siebel.

The BPEL service 'UpdateSalesOrderSiebelABCSImpl' is configured with 'transaction' attribute of 'required' and 'oneWayDeliveryPolicy' of 'sync'. This ensures that BPEL processing and the Siebel invocation participates in the same transaction that was started by the Worker thread of the Resequencer.

More information about this flow is documented at
http://docs.oracle.com/cd/E38316_01/doc.114/e37675/com_olm_so_update_impl.htm

Performance under load scenarios

In large load scenarios, if on an average 10,000 new orders an processed every hour for 7,000 new customers, then there are 7,000 Resequencer groups and since there are 9 updates for every order, there are 90,000 Resequencer messages created every hour. In 8 business hours of the day this amounts to approximately, 720,000 Resequencer messages and 56,000 Resequencer groups. This Oracle AIA flow is used by customers under much larger loads than mentioned above.

Under such loads, there will be an impact on the throughput. However, with the approaches of auto-deletion of Resequencer messages, purging of groups and Resequencer thread tuning as discussed in the Performance tuning section above, the throughput was significantly improved. When using clustered deployments and large loads, the Resequencer transparently manages load distribution under failover scenarios ensuring that no single node is affected by a higher load.

A note on Single-threaded vs. Multi-threaded JMS Adapter

It must be noted that in figure 30 above, the JMS adapter is shown as single-threaded since a FIFO Resequencer was used. The single thread should deliver messages to the Resequencer fast enough to keep the Resequencer busy as there may be hundreds of worker threads across the deployment. Sometimes, PIP customers increase the JMS Adapter threads to improve throughput. Increase in threads can also happen due to clustering for high availability, which was discussed earlier in this paper.

As per FIFO Resequencer best practices that were discussed earlier in this paper, invoking a FIFO Resequencer in a multi-threaded pattern can result in out of sequence processing (Refer Figure 10). However, in this case since the updates from OSM are known to arrive in an interval of approximately 5 seconds, the chances of out of order processing is negligible.

However this assumption can break under the following scenarios

- If OSM is configured to send more frequent updates
- If due to some OSM connectivity issues, messages are backed up in OSM and delivered to the JMS Queue in a bulk fashion
- If the number of threads are increased drastically (remember, as discussed earlier, clustering increases the number of JMS adapter threads very quickly. 10 node cluster with each node configured with 5 adapter thread results in 500 adapter threads)

It is therefore recommended to keep the adapter threads down to 1 or as low as possible.

Use case – Multiple Resequencers

Assume a scenario where product/item information is synchronized between a source system and multiple target systems. A message from the source system has to be synchronized with one or more target systems depending on the content in the message. Using AIA Foundation Pack concepts, this system is designed at high level using canonical model as follows.

In this system, the item message sent from the source system through a web service invocation is validated and transformed to an AIA based canonical message by the SyncItemListSourceReqABCSImpl BPEL service. This service invokes the Item Enterprise Business Service(EBS) mediator service using the canonical message. This mediator introspects the canonical message and invokes the appropriate target system based on the routing rule defined in the mediator. For each target system there is another BPEL service which transforms the canonical back to the appropriate target application message format. Note that the mediator may need to route the same product information to more than one target system.

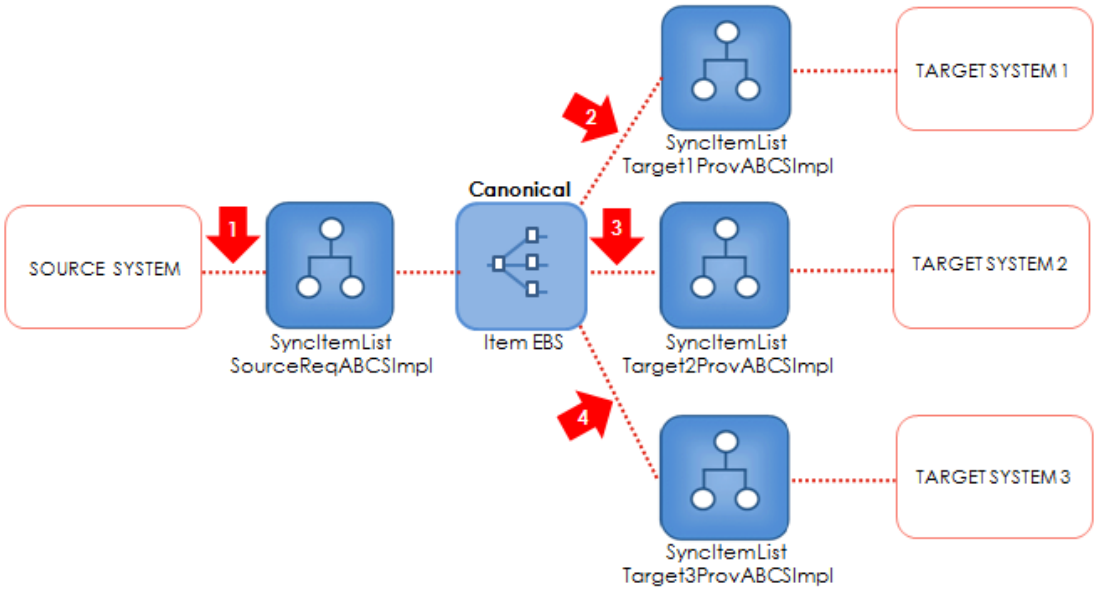

Figure 31: Multiple Resequencer Scenarios

In this case the business states that the following requirements

- Data Integrity – Updates should be made in the same sequence as sent by the source system. Errors should cause the system to halt instead processing further records.
- Guaranteed delivery
- The target systems can accept only one update at a time.

It is obvious that a Resequencer is required but the question is where the Resequencer should be ideally introduced. This paper will not conclude on a design but illustrate a thought process giving various options to consider and their advantages and disadvantages

Option 1:

If a FIFO Resequencer is introduced in point 1 shown in the figure and nowhere else, then incoming messages will be correctly re-sequenced and processed. However to continue single-threaded processing, the design also requires that ItemEBS mediator use sequential routing rules (since parallel routing rules will introduce new threads) and also requires the BPEL processes to execute in the same worker thread using the oneWayDeliveryPolicy parameter discussed earlier.

The downside of this design would be that each worker thread takes a long time to execute. Since sequential routing rules are being used, for a given message, the same thread is used for executing all BPEL processes, as well as invoking each target (assuming that more than one target is reached). If the BPEL processing is very light, if the messages are small and few and if the BPEL process simply posts the messages to target applications database tables or JMS Queues, then this option may work reasonably well since these worker threads are held for a lesser time and the tables/queues maintain the order as well. If this is not the case, then as discussed in the tuning section, such long processing will introduce significant backlog in Resequencer processing. This can also cause timeouts.

Another factor to consider is that if one target system or Target BPEL process errors out, then the entire transaction including encompassing all targets will need to roll back. This may not be desired.

Option 2:

If, instead of position 1, three Resequencers are added one each at 2, 3 and 4 positions as shown in figure 18, then it would solve some of the short comings faced by option 1. In this case, even though a sequential routing rule is used in the ItemEBS, the messages are simply committed to each Resequencer eventually. The messages are then processed individually independent of each other giving a better operational model as well as improved performance (quicker worker thread turnaround). Since the messages are stored in the Resequencer tables, they are guaranteed to be delivered. Each Resequencer will also process messages sequentially and halt the processing in case of errors, which is desired. The Resequencers can also be tuned independently based on the characteristics of their targets.

The disadvantage of this approach is that it cannot guarantee that the Resequencers will receive the messages in the same order in which they were delivered by the source system. This is mainly because of the multi-threaded nature of the Source BPEL component. In an HA setup, there is a higher chance of messages being processed out of order. Additionally if an Item update fails in the Source ABCS BPEL service, then a subsequent update for the same Item will still be processed by the Source ABCS BPEL process. The Resequencers at 2, 3 and 4 will be oblivious to this error.

Option 3:

In this option, Resequencers are introduced in all 4 positions 1 through 4. This is an improvement over option 2 because it ensures that the incoming messages are sequenced and single threaded even before they reach the ItemEBS. Also, if a message fails in the Source ABCS BPEL process then all messages for that item are entirely blocked.

This seems as a good option, although four Resequencers can introduce an increased cost of operations (fault management and data purging) and also requires more performance tuning. Under

heavy load and large sized deployments, these factors will be more pronounced and so will need to be carefully planned.

These are some of the design options. There could more design options such as using Best Effort Resequencer, Oracle Weblogic JMS UOO, etc.

Comparison of Resequencer with Weblogic JMS UOO and UOW

Weblogic JMS Unit of Order (UOO) and Unit of Work (UOW) features provide message ordering capabilities comparable to that offered by the Resequencer. The official documentation for UOO and UOW is available at http://docs.oracle.com/cd/E13222_01/wls/docs103/jms/uoo.html and http://docs.oracle.com/cd/E15051_01/wls/docs103/jms/uow.html

This section will not focus on describing these features and it is assumed that the reader understands the basics of UOO and UOW as described in the links above. The focus will instead be on some of the similarities and differences when comparing the Resequencer with Weblogic JMS UOO/UOW and discuss use cases where UOO/UOW may be better suited.

At the outset, UOO and UOW are Weblogic JMS related. Therefore its applicability is related only to integration scenarios that involve JMS interaction. However, if the design permits, JMS Queues may be introduced specifically into a SOA Suite integration flow, to leverage these features. The Resequencers on the other hand are applicable only when SOA Suite is employed. It is an integral part of the SOA Infra and so works seamlessly within a SOA composite. For example, in the 'Multiple Resequencer' use case discussed earlier, it is logical to use a Resequencer instead of a JMS Queue/Topic when sequencing between multiple BPEL processes. In this case, using Mediator Resequencer can provide better performance, better error handling and monitoring abilities.

In addition to the above, a mediator can support multiple operations some with resequencing and some without. This reduces the deployment foot print when compared to configuring and maintaining multiple Weblogic JMS Queues/Topics. Mediators also allow for transformation during resequencing, whereas when using JMS, additional components may be required to achieve transformations.

UOO vs. FIFO Resequencer

UOO is comparable to a FIFO Resequencer. In UOO, messages belonging to the same UOO name will be processed in a single threaded fashion based on the time of arrival. Only one message of a particular UOO is made available to the JMS consumer at a given time. However the UOO name has to be configured at the connection factory level or it has to be set programmatically by the calling application. In the Sales Order processing example, it would require additional logic at the Siebel end to programmatically set Order ID as the UOO name, before producing the message to the JMS Queue. While the approach of UOO has its own benefits and applications in some scenarios, in this case of Sales order processing, a Resequencer is better suited.

Another difference is that, when a message is consumed and it fails in downstream processing (with an acknowledgement), then UOO doesn't block the subsequent message from being consumed. This is

different from FIFO Resequencer. In the Queue vs. Resequencer anti-pattern it was described that when using Group ID as Account ID, one order failure unnecessarily blocks subsequent orders. For such a scenario UOO is better suited since it does not block subsequent messages.

Another factor to consider is that Unit-of-Order will override JMS sort criteria, priority, or filters which may be otherwise required in the design. In such cases one design option would be to not use UOO and instead use a Resequencer. For example, the AIA use cases in http://docs.oracle.com/cd/E24010_01/doc.111/e22651/appendix_ofm1.htm provide use cases where both JMS priorities and Mediator Sequencing are used together. In this case, the JMS (without UOO) ensures that the Sales Orders of higher priority are delivered prior to the earlier to the orders with lower priority while Resequencers ensure that updates to these orders are delivered in sequence.

UOW vs. Standard Resequencer

UOW allows defining a 'Sequence ID' (in addition to a UOW name) that defines the sequence in which the messages are to be assembled. Similar to a Standard Resequencer, the arrival times don't matter but instead the messages are strictly sequenced based on the Sequence ID.

However there are many differences that make each suited for different scenarios. UOW assembles a complete set of ordered message that may arrive out of sequence through one or more producers. Once the message set is formed it is made ready for consumption in one single transaction. Unlike Resequencers, UOW requires an 'end' message to be defined. Until all messages including the end message arrive in the queue, no message is delivered to the consumer. If the consumer errors during downstream processing, then the entire batch is rolled back. **In contrast to this batching behavior, a Resequencer treats messages as a stream and messages are processed as and when partial sequences arrive**.

The link http://docs.oracle.com/cd/E15051_01/wls/docs103/jms/uow.html explains an online retailer scenario where the number of messages is known. In such cases UOW is useful as it ensures that a partial order is not invoiced. In the case of Standard Resequencer scenario discussed earlier, when sending multiple order versions, there is no way to know the total number of order verions ahead of time. Also since Order versions are expected to be processed as and when they occur, a Standard Resequencer is a better fit.

UOW also allows users to specify the Sequence ID which may be handy in scenarios that involve more than one source application producing messages. UOW is also useful in split and aggregate scenarios. In the Online retailer example, the order may have arrived as single order with multiple lines and these lines could've been split and injected with different sequences ids and aggregated later. Such scenarios are not possible in a Standard Resequencer.

Similar to a Standard Resequencer, UOW also times out waiting for sequences/end message. The timeout is configured as the time elapsed since the first message of the UOW arrived. Exception Policies can be set to redirect timed out messages to error queues. The time out should not be set to high value. Since messages are not delivered until a sequence is formed, UOW can have poor performance when having a large back up of messages. If the messages are large, the performance can become worse due to memory constraints. Additionally all the messages are delivered to the consumer

in one batch which could suddenly increase memory consumptions. Resequencers do not suffer from this problem as all messages and sequencing metadata are stored in the database.

Summary

Message resequencing requirements are inevitable in many large integration scenarios. When implementing a SOA based integration, an Oracle Mediator Resequencer can be introduced easily in a declarative fashion, anywhere in a complex message orchestration, without requiring any custom development effort. This paper aimed at providing some of the design considerations and configuration best practices when using the Resequencer.

The Resequencer is robust and is backed by the market leading Fusion Middleware Infrastructure, development tools and monitoring tools. Several Oracle customers use Oracle the Resequencer from small to very large deployments and Oracle continues to invest in Oracle Mediator for future releases.

Message Sequencing using Oracle Mediator Resequencer
September 2013
Author: Arvind Srinivasamoorthy
Contributing Author: Shreenidhi Raghuram

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment

Hardware and Software, Engineered to Work Together