JPMQ—An Advanced Persistent Message Queuing Service

Hans-Peter Steiert and Jürgen Zimmermann

Department of Computer Science, University of Kaiserslautern, P. O. Box 3049
D-67653 Kaiserslautern, Germany
{steiert, jnzimmer}@informatik.uni-kl.de
http://www.uni-kl.de/AG-Haerder/

Abstract. Message oriented middleware (MOM) services, based on the concept of persistent queues, are widely used for reliable asynchronous communication to deal with the disadvantages of the closely coupled communication model. This paper introduces some new functionality for queuing services (QS) in order to make them more powerful for the need of modern application programs and presents JPMQ, a sample implementation of these concepts. As such, JPMQ provides an object-oriented type system for messages, dynamic selection and ordering of messages as well as content-based priorities. Moreover, JPMQ offers transactional concepts in order to cope with concurrent access and failure recovery.

1 Introduction

Large-scale business applications are often comprised of autonomous, distributed programs, each of which performs a certain piece of work of a (more complex) task. In such domains, reliable communication is one of the primary necessities for correct task processing because the current state-of-progress has to be communicated between the programs. A closely coupled communication model based on a distributed two-phase commit protocol ([9]) allows programs to exchange data (i. e. the task's state-of-progress) in a reliable way. Usually a directed flow of information can be observed. Therefore, communication is sufficiently reliable/stable if the sending program can be 'sure' that the information sent will be received by the 'next-step' program.

This is a major issue addressed by so-called queuing services (QS), also known as message oriented middleware (MOM, [14]). These systems manage messages—as units of data to be exchanged—by using queues. After a client program has stored a message in a queue, it may continue processing without having to wait for the 'next-step' program to accept the message. Once a message is queued, the QS guarantees delivery, even when the target system is not reachable (either intentionally or due to failure) at the time of queuing ([2]). As for every base service, a seamless integration with application development and easy operation are necessary.

We want to address these topics by presenting a value-added QS, called JPMQ (*Java Persistent Message Queues*, [12], [18]). It provides semantically richer functionality, as for example queries on the contents of queues. Since this functionality is integrated into the enqueue and dequeue operations, application development can easily take advantage of these features.

This paper is organized in the following manner. In the next section, we give a short introduction into persistent queuing and the basic concepts realized in current queuing systems. Then we introduce some concepts we call "advanced queuing functionality", which go beyond the queuing facilities supported by current QS. Sect. 4 outlines our queuing system JPMQ, followed by the description of its implementation. We finish up with conclusions and a look into future work.

2 Persistent Queuing

Persistent queuing is a widely used technique for asynchronous communication between application programs. While messaging and synchronous communication can only be applied if the interacting programs run at the same time, the usage of a QS for communication relaxes this restriction.

The use of a QS simplifies the development of applications with asynchronous peer-to-peer communication, because it provides a reliable message buffer on the basis of queues. To offer an easy-to-use programming scheme, a QS integrates several concepts, namely those of messages, queues, and—most important in respect to quality of service concerns—an appropriate transaction semantics.

Queues can be considered from different perspectives, namely from the point of view of the QS and from that of the application programmer.

For the QS, queues are containers for arbitrary messages, which are under its control. Therefore, the QS takes care of all defined queues and their associated attributes. Queues accept an application program's storage and retrieval requests for messages which are commonly known as *enqueue* and *dequeue* operations.

From an application programmer's point of view, queues provide a helpful abstraction for distributed application development that uses programs in producer-consumer relationships. As such, producer and consumer do not have to know each other since the queue abstracts from a direct relationship. For a producer, a queue is a destination for its output data, while for a consumer it is a source of its input.

As we have already mentioned, the basic operations on queues are *enqueue* and *dequeue*. The enqueue operation is used by an application program to store a message into a specific queue. When an enqueue operation is processed successfully, the QS takes the message and guarantees its permanent storage. The counterpart of the enqueue operation is the dequeue operation. An application program uses the dequeue operation to request the delivery of a message from a specified queue. A successfully terminated dequeue operation releases the QS from controlling the delivered message. It may be possible to dequeue more than one message at once from the queue using a so-called "multiple dequeue" operation.

In order to enable application programs to differentiate messages according to their importance, a QS can allow explicit assignment of priority values along with the enqueue operation. For messages with a priority value, a specialized option of the dequeue operation exists which removes messages from the queue in ascending or descending order of these priority values. Messages enqueued without any priority value are considered to be of a static priority.

Transactional concepts for operations upon queues are desirable for concurrent access and failure recovery ([9]). In the context of a QS, we will reconsider the ACID transaction paradigm ([10]) which has proven to be the most appropriate one.

Obviously the *durability* property must be provided by each QS. It ensures that enqueued messages do not vanish (until dequeue) and that dequeued messages are no longer accessible.

Atomicity is desirable in the sense that the entire set of dequeue and enqueue operations made during one transaction is performed successfully or none of them are. The consequence for the QS is, that effects of unsuccessfully terminated transactions (e. g. due to a program crash) will not show up.

Isolated execution is a means of coping with the requirement of atomicity. If all operations that an application program performs on queues are isolated from (i. e. not visible to) operations of other application programs, then those operations can be rolled back without affecting another application program's work.

With regard to *consistency*, the possibility of defining some consistency rules for messages is desirable. The simplest consistency requirement is surely that enqueued messages are not altered as long as the QS is responsible for them. Furthermore, some consistency rules could be defined for queued messages, e. g., to prevent the insertion of wrong data.

3 Advanced Queuing Functionality

The actual reason for the development of JPMQ was the need for so-called *advanced queuing functionality*, since rudimentary services such as enqueue or dequeue operations are already provided by today's products ([1], [4], [7]).

For the support of this advanced functionality, we will introduce complex message types which allow for the specification and control of the internal structure of messages to be exchanged. Starting from those complex messages, we propose several extensions based on the message contents: automatic priority assignment of enqueued messages, application-specific dequeue ordering and selective dequeuing. Another point we address are correctness issues guaranteed by the QS for queues in a multi-producer, multi-consumer environment. In the following, we want to discuss each of these extensions in more detail.

3.1 Complex Message Types

As already mentioned, a QS will be a useful middleware service for building distributed applications. Today's business applications are developed using object-oriented design methods ([3], [11]). The use of a semantically rich object model

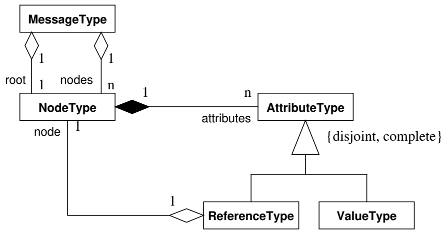

Fig. 1. Metamodel of the JPMQ type system (UML notation)

in the design phase is often reflected by the usage of structured data in the implementation. Hence, an adequate representation of data transferred by messages is not a flat byte stream but a structured data type. This leads to the conclusion that the QS should be aware of the structure of the messages stored. In JPMQ, this is obtained by strict typing, which means that a message type is associated with every message. Thus, JPMQ has to provide a type system and a mechanism to define message types. The main design objective for the type system is to make it simple to use but flexible enough to define complex message types for almost every application domain.

JPMQ message types are defined using a three level hierarchy, which directly reflects the different abstraction levels. The topmost concept is that of a *message type* used for the strict typing of messages. A message type defines the unit of data exchange and has a service-wide unique identifier. As messages are complex structures, a message type consists of one or more *node types*. These node types group together a set of *attributes* that describe the different instances of this node type. A node type carries an identifier which must be unique within the message type. The attributes of a node type itself are again uniquely identified by their name, and can hold either *values* or *references*. Values are of simple type (e. g., strings, boolean or numerical values) whereas references represent relationships between different nodes. As a message type may contain more than one node type, one of them must be designated as the message type's *root node*. An instance of this node type becomes the only entry point (root) from which the remaining parts (nodes) of the message can be reached. A metamodel of our type system is shown in Fig. 1, an example message type definition in Fig. 2.

3.2 Automatic Priority Assignment

As JPMQ is aware of a message's internal structure, it can easily access the values stored within a message. A rather simple feature of JPMQ is automatic priority calculation and assignment for messages enqueued without an explicitly given priority value (see Sect. 2).

The priority value calculation is done by the means of a specific "priority calculation function" (PCF) which has to be specified in an OQL-like query language (see ODMG standard [5]). The PCF is evaluated while enqueuing the message and delivers a single value of type "real", which is taken as the message's

```
DEFINE MESSAGETYPE OrderMessage;
  NODE order;
    id: IDENT; date: DATE; lines: REFERENCE TO order_line;
  END;
  NODE order_line;
     line_no: INTEGER; amout: INTEGER; price: REAL; item: REFERENCE TO item;
  END;
  NODE item;
    id: IDENT; name: STRING; ean: INTEGER;
  END;
  ROOT order;
END;
```

Fig. 2. Example DDL statement for message type definition

priority. A PCF is always associated with a message type and a queue. The conjunction with a message type is obvious, since the query itself is tailored to the structure of that specific message type. The association with a queue allows for more freedom in defining special policies for different queues which accept the same message type.

3.3 Application-specific dequeue ordering

Besides the rather static ordering of messages based on priority values assigned, JPMQ allows for application-specific dequeue ordering using a content-based sorting criterion. By this means, a reading application program can dynamically specify the order in which messages should be dequeued. For that purpose, we introduce a specialized dequeue operation that accepts a content-based "ordering function" (OF) which is conceptually comparable to a PCF except for the time of evaluation. While the PCF is statically defined once and evaluated along with the enqueue operation, the OF is dynamically issued and evaluated along with the dequeue operation to assign priority values to every message of the given message type residing in the queue used. In contrast to a PCF, an OF can deliver an ordered set of base types which is used to order the messages (see Sect. 5.3). Because PCFs make it possible to order all messages in a queue, for every message type we should get the same result type. As OFs are restricted to one message type, their usage can be simplified by allowing ordered sets as comparison values.

3.4 Selective Dequeue Operations

Besides dequeuing messages in a specific order (using PCFs or OFs), application programs often have to work exclusively with messages which satisfy certain predicates.

If only an ordinary dequeue operation is available, an application program has to dequeue all messages, select the qualifying ones and re-enqueue those that are not to be processed. Because all this has to be done within a single transaction (in order to avoid loss of messages), non-qualifying but read messages are not accessible to other application programs until commit, so concurrency is reduced in most cases. Thus, a better solution would be to enable the QS by appropriate means to perform the selection 'nearby the queue', i. e., within the dequeue operation.

Providing this feature in JPMQ was no problem, because the message structure may serve as a foundation for the specification and evaluation of selection predicates. Thus, we introduce the "selective dequeue" operation. It accepts a "selection predicate" (SP), which (again) is specified in an OQL-like style. The SP is defined upon a concrete message type and evaluated during the dequeue operation on the messages of that type which are stored within the addressed queue. Only those messages which are included in the result set of the SP are delivered.

As a matter of fact, both the selective and sorting dequeue operations affect messages of a single type. While selection is evaluated on every single message, the ordering is done on a set of messages, so they can both be applied within a single dequeue operation which we call "selective ordering dequeue", as shown in the following example:

```
orders = outstandingOrdersQueue.dequeue
          ( "ORDER",
            no_of_orders,
            "count ( select ol.items from mt_OrderType om, om.root o, o.order_line ol )",
            "select om from mt_OrderType om, om.root o where o.name = 'Smith'" );
```

3.5 Isolation issues of JPMQ in a multi-user environment

So far, we didn't consider concurrency w. r. t. enqueuing and dequeuing messages. We now want to address the related problems by introducing different isolation levels. Programs enqueuing messages are called writers, those dequeuing message are called readers from now on.

JPMQ demands a queue to be in one of the four isolation levels "none", "ordered", "blockwise", or "ordered blockwise". These different levels stem from two independent factors, which determine the degree of the desired isolation. One factor is the application program from which messages originate and the other is the actual transaction (with visible transaction borders) used for enqueuing. The simplest (or lowest) isolation level ("none") means that there is no isolation between dequeuing transactions.

Let us, first of all, consider the case that there is just one reader. Without any ordering, he gets the messages in the order in which they are enqueued, so that we have a FIFO processing. It might happen that the reader gets messages enqueued by different writers in an intermixed way. Let us now imagine that there are two readers. Seen from the queue, this could lead to an intermixture of dequeue requests stemming from *Reader1* and *Reader2*, e. g., d1[0], d1[1], d2[0], d1[2], d2[1], d1[3] (of Fig. 3, left). The messages in this example are labelled with a number identifying the writer and a letter used as a message identifier.

If there is an abort of *Reader2* (as illustrated in Fig. 3, left) the QS might react by delivering "1/c" as result of the next dequeue operation (d1[3]) of *Reader1*, leading to a non-FIFO order of messages dequeued by *Reader1*. This is exactly

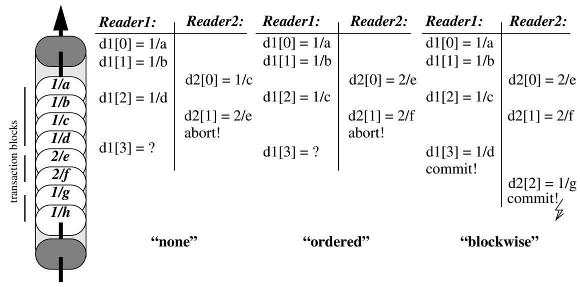

Fig. 3. Effects of different Isolation Levels

what might happen when using isolation level "none". Hence, it is only useful, if there are no dependencies existing between different messages of the same writer.

To avoid such inconsistencies, we can isolate the two readers on the basis of the writers in such a way, that only one reader has access to the messages of one writer at a time, but may access messages of several writers. This level of isolation is called "ordered", which means that JPMQ guarantees a FIFO order for messages of one writer. Still, messages from different writers could be dequeued in an intermixed way, e. g. the result of d1[3] may be message "2/e" (Fig. 3, center), but *Reader2* is not able to receive any message from *Writer1* before *Reader1* has committed.

Up to isolation level "ordered" messages of different writers could be dequeued in in arbitrary intermixture. This may be sufficient in some cases, but for the realization of protocols with strong inter-message dependencies, neither "none" nor "ordered" is strong enough. Here, it makes more sense to take transaction boundaries into account: Only one reader gets exclusive access to messages of one (writing) transaction and has to dequeue all messages of this transaction. This means, the reader is only allowed to commit successfully if there are no more messages left from the writing transaction. The corresponding isolation level is called "blockwise" (Fig. 3, right).

Comparing the isolation levels "ordered" and "blockwise", we see that they address different problems. "Ordered" reserves all messages of a writer for the reader that accesses the first of his enqueued messages without taking (enqueuing) transactions into account. "Blockwise" only reserves the messages of a single enqueuing transaction, but imposes that the reader is aware of the transaction boundaries. So, "ordered" assures the writer's FIFO order, whereas "blockwise" reflects transactions. As such, it is a logical consequence that another isolation level exists, which combines those two concepts. This level is called "ordered blockwise" and has the following semantics: The first time a reader dequeues a message of a writer, it is guaranteed that this is the oldest message of that writer (that is currently in the queue) and, at the same time, exclusive access to all messages of this writer is granted. Similar to "blockwise", the reader can dequeue only messages of transactions of this writer in the order in which they are

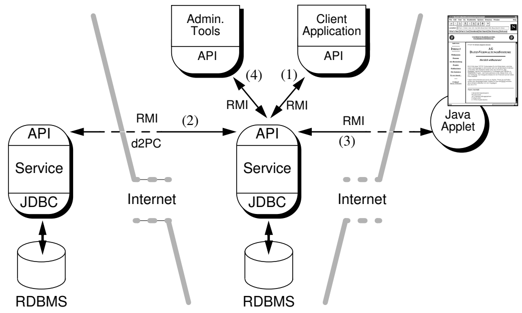

Fig. 4. External Architecure

enqueued. Moreover, a reader transaction must commit at one of the transaction boundaries which stem from the writers' transaction commit.

4 JPMQ System Architecture

In the following sections we will take a closer look at our implementation of JPMQ. The QS was designed as a middleware service used in a distributed computing infrastructure. We will explain the usage of JPMQ in such an environment and introduce its interfaces (Sect. 4.1). After that, we will examine the main components of JPMQ and how they cooperate to fulfil the requested services (Sect. 4.2). A more detailed description of implementation details is the main topic of Sect. 5.

4.1 Overall architecture

In this section we will use the notion of "nodes" to differentiate between two QSs. The term *local node* is used as an alias for the JPMQ service to which an application program is currently connected. Accordingly, a *remote node* is a QS instance not known to the application program. Remote nodes may reside somewhere in the internet, in the local domain as well as on another continent.

JPMQ functionality may be claimed using one of four different interfaces (Fig. 4). The standard application programming interface (API, (1)) enables local application programs to access JPMQ. For communication between different instances of JPMQ, e. g. forwarding messages from a local to a remote node, a service-to-service interface (2) exists, which enables two JPMQ services to exchange messages. A distributed two-phase-commit protocol is used to ensure that no messages are lost. An applet-enabled interface (3) is provided in order to support lightweight clients, as for instance Web-clients or mobile computers. Electronic commerce will be a typical application using this kind of interface. In addition to the described interfaces, a fourth one is needed to create message types and queues in order to change default settings and to do maintenance (4).

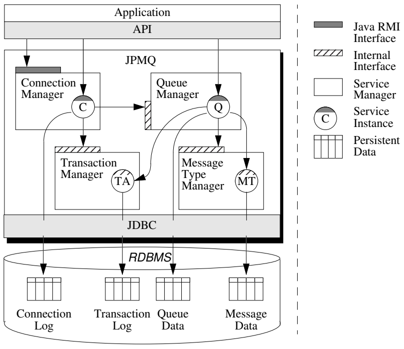

Fig. 5. Internal Architecure

4.2 Inside JPMQ

Let us now have a closer look inside JPMQ. Fig. 5 gives an overview of the main components, which cooperate together in order to achieve requested operations. We are exploiting the features of the Java-RMI mechanism for all client/server communication. Java-RMI hides the aspects of distributed object computing to the application developer ([17]).

First of all, an application program has to connect to a JPMQ node. Then it may open some queues in order to enqueue or dequeue messages. Before issuing the first enqueue or dequeue operation a transaction has to be started.

The *connection manager* handles the incoming connect request to establish a connection to this node. This is achieved by creating a new connection object and passing it back to the application. This connection object privides the functionality to open or close queues. Requests to open a queue are forwarded to the *queue manager*, which is responsible for all queues managed by this JPMQ instance. The queue manager creates a new queue object, which is a transient representative of the persistent queue. As a result of the 'open queue' request, the corresponding queue object is passed to the applications.

Operations on queues are not allowed outside the scope of a running transaction. Transaction control is done by transaction objects. They are created by the *transaction manager* if a 'begin-of-transaction' operation is issued by the application. These requests are sent to the connection object and forwarded to the transaction manager. Whenever an enqueue or dequeue operation is performed by a queue object, the transaction object is notified and a log entry is written. In case of a system failure, these logs are taken in order to perform crash recovery. The implementation details of transaction control are covered in Sect. 5.2.

JPMQ is used to manage persistent queues with complex message types. A message consists of a header and a body. While the body contains application data, the header consists of information used by the queuing service itself. The header is always made of the same structure. Therefore, writing or reading the header information is done by the queue service. Because various message types have different internal structures, a specialized message type service object, which is managed by the *message type manager*, exists for every message type. Reading and writing a message's application data is done by the corresponding message type service object.

5 Implementation

This section deals with implementation details of JPMQ. One very important design decision was to use an RDBMS as a storage layer. The advantages of using a RDBMS are discussed in Sect. 5.1. However, we do not consider the RDBMS transaction mechanism as sufficient for our QS. As already mentioned in Sect. 4.2, we decided to use our own transaction control. How it is based upon the ACID transactions provided by the RDBMS will be the topic of Sect. 5.2. Furthermore, we will discuss how JPMQ handles structured message types and how selective or ordering dequeue operations can utilize the query capabilities of RDBMSs (Sect. 5.3). In order to facilitate development, the object-oriented programming language Java is used for the implementation of JPMQ. Hence, we present our experiences using Java in Sect. 5.4.

5.1 The Storage Layer or Why to use an RDBMS?

The advanced features we have already described did not cover one of the main facilities that any QS must handle: the reliable and persistent storage of messages. While messages can be persistently stored in various ways, we have chosen the proven technology of RDBMSs. This was done for several reasons, some of which stem from the capabilities of RDBMSs, and some of which allow us an easier and more powerful realization of the advanced features of JPMQ.

One of the reasons for a decision in favour of an RDBMS are the transaction properties, in our case, primarily duration and atomicity. If complete messages (with all nodes and their references) are stored in a database using a single database transaction, the RDBMS guarantees that, after a successful commit of that transaction, all the data is stored in a durable manner. Also, the atomicity property is useful, as a message is either stored completely or not, but not partially. Moreover, the RDBMS takes responsibility for data recovery actions to be taken after a system crash, so that JPMQ is grounded on a valid data basis. In Sect. 5.2, we describe how this is used as a bottom layer to build up our advanced transaction control.

An RDBMSs as the lowest level of data storage does not only bring us the advantage of ACID properties for single enqueue/dequeue operations, but also supports us in bridging the heterogeneity of the application domain in which JPMQ resides. Since the message contents are stored in attributes of database records (see Sect. 5.3), the RDBMS rather than the JPMQ is responsible for the physical representation of the values stored. Moreover, the use of SQL relieves us from becoming dependent on a specific RDBMS as long as it supports an appropriate implementation of the SQL standard ([15], [13]). This enables our system to run on top of almost every RDBMS which supports SQL if a suitable

JDBC driver is available. The JDBC interface is used to access the database as can be seen in Fig. 5.

One last, but not less important, reason for the usage of RDBMSs are the query facilities of relational database technology in the form of SQL. This allows us to utilize the RDBMS's query facilities to do value-based selection and ordering of messages for the advanced dequeue operations on certain message types.

5.2 Transaction Management

JPMQ allows grouping a set of enqueue and dequeue operations together as a transaction. Either all operations within this transaction are successfully done or none of them are. Additionally, a queue can be defined to ensure different levels w. r. t. the order of the messages inside the queue, even if several applications access the queue at the same time. As we have seen before, some of these properties are similar to database transactions.

Mapping every JPMQ transaction onto a database transaction would have been an easy way to implement JPMQ's transaction management. Unfortunately, this would lead to less concurrency, because every operation has to write data in tables holding administrative data, e. g. locking and logging information. Therefore, JPMQ maps every single queue operation onto a database transaction of their own, inheriting the durability properties of database transactions. This approach allows high concurrency in accessing the queue, but requires further steps to ensure atomicity and isolation. Before examining logging and recovery in more detail, we will discuss in the next section how JPMQ ensures isolation of concurrent readers and writers.

Isolation. JPMQ implements isolation by making messages invisible to transactions which are not allowed to access them. A flag, called *visibility state flag* (VSF), is used to hide messages. If a message is visible to all transactions, the VSF is set to '$\star$', otherwise it is set to the identifier of the transaction with exclusive access to the message.

The VSF is stored together with the *message identifier* (MID) and the *log state flag* (LSF) in the administrative data belonging to every message. This triple (MID, LF, VSF) is used to compose the message's state. The MID is built from the three parts S, T and M, whereby S is the identifier of the writer, e. g. the writing connection. T is the identifier of the transaction which enqueued this message. It is unique inside the connection S. The third part, M, is a unique identifier used to determine the message itself. The concatenation of S, T, and M—separated by '/'—forms the MID. We will refer to this representation of the MID when explaining the isolation protocols of the different isolation levels.

Fig. 6, left, shows the different message states which may occur while a message stays physically inside a queue with isolation level "none". After a writer 'a' has enqueued message 'n' in transaction 'a/1', the VSF is set to 'a/1' (message state $(\mathrm{a}/1/1, \mathrm{e}, \mathrm{a}/1)$). Now the message is only visible to operations performed

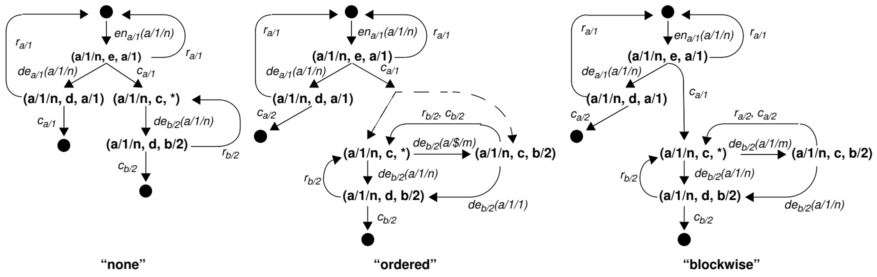

Fig. 6. Visibility Graphs

by transaction 'a/1'. If writer 'a' commits the transaction, the VSF is set to '$\star$' making the message visible to all other transactions (message state (a/1/1, c, $\star$)). In isolation level "none", no other operation will affect the VSF except for a rollback ($r_{a/1}$), which is described as part of the recovery.

If isolation level "ordered" or "blockwise ordered" is used, the message state diagram is more complex, because the VSF may also be affected by additional operations. First, the commit of transaction 'a/1' may result either in state (a/1/n, c, $\star$) or (a/1/n, d, b/2). Second, a dequeue operation will affect the VSF, even if another message is read from the queue. Both result from the need to make all messages of a single writer invisible to other transactions, if transaction 'b/2' reads the first message of writer 'a' (Fig. 6, center).

Assume transaction '2' of reader 'b' reads message 'a/2/m'. In this case, the VSF of all messages 'a/\$/\$' is set to 'b/2', making these messages invisible to transactions other than 'b/2'. Here '\$' may be any T or M in 'S/T/M'. If transaction 'a/1' commits after 'b/2' has read a message 'a/\$/\$' from the queue, the state of the message 'a/1/n' enqueued by 'a/1' is (a/1/n, c, b/2) instead of (a/1/n, c, $\star$).

The isolation protocol used in isolation level "blockwise" differs only a little bit from the one examined above. If a message is read, then all messages of the same transaction are invisible until the reading transaction commits (Fig. 6, right).

When implementing the message state diagrams presented in this section we exploit the set-oriented operations of the RDBMS used. Every arc of the diagram can be implemented using one query. The state of all messages is updated to the new state, by using the old state as the qualification criterion in the WHERE clause of the update (or delete) operation.

Notice that the order itself is not ensured by the isolation protocol. The messages are ordered on dequeue by using a sufficient order criterion along with the queries. In isolation level "none" or "ordered", all messages are sorted using their timestamp. If the isolation level is set to "blockwise", JPMQ orders the messages using the 'S/T' of the MID combined with a timestamp as an order criterion in the appropriate database queries. Hence, the levels "ordered" and "ordered blockwise" are reflected by the same message state diagram.

Logging and Recovery. As mentioned before, JPMQ has to track every queue operation. The corresponding log records are stored in different database tables. Because the log records are written in the same database transaction as the corresponding queue operation, it is ensured that the log is consistent with the application data, i. e., the messages written to the queue. After a crash occurs, the log is analysed. If there are operations of uncommitted transactions pending, undo recovery is necessary. There is no need for any redo recovery, because all operations of a queue transaction are reflected in the database after their commit. As messages cannot be changed while they are held under control of the QS and dequeued messages remain in the database until commit, there is no need for logging before-images. Instead, JPMQ writes log entries only for the operations performed upon the queue during a transaction into the log tables. Beside the VSF, the message state also carries a *log state flag* (LSF). This flag is used to mark the state of the message regarding to the log entries.

Take the following example. Assume the isolation level is set to "none" (Fig. 6, left). After a message is enqueued by transaction 'a/1', the LSF of this message is set to 'e'. Now, the system crashes before committing 'a/1'. So, all messages written by transaction 'a/1' must be removed from the queue. This means that all messages in message state (a/1/$, e, a/1) are deleted, an operation easily performed using the set-oriented operations of the RDBMS. Undo of dequeue operations is also easy. All messages dequeued by transaction 'b/2' are in message state (a/$/$, d, b/2). Hence, undo is done by updating the message state as shown in (Fig. 6, left) to the new state (a/$/$, c, $\star$). As we can see in the different message state diagrams some special actions have to be performed if isolation levels other than "none" are used or messages are enqueued and dequeued by the same transaction. For example, the VSF of messages not enqueued or dequeued by the transaction may have changed in order to ensure correct isolation. JPMQ has to undo these changes, too. In the state diagrams an 'r' marks all arcs which belong to undo operations. By examining the log entries, JPMQ decides which of these state transitions must be performed and how the set-oriented operations of the storage layer can be utilized.

5.3 Complex message types and query evaluation

In this section, we will describe how JPMQ exploits the data management facilities in order to store message types and to evaluate queries. For this reason, we need a mapping of the complex message types onto the relational data model, and a transformation of the OQL-like queries of JPMQ to SQL queries.

Let us first examine the mapping of the complex message types. Each message consists of a header and a body. The header holds the administrative information regarding the message. This is reflected by a special node added to the message type by JPMQ. It is always used as the root (header) of a message with a structure uniform for all message types and includes administrative data such as the priority value of the message, the lock and visibility flags, the message type name, and a timestamp of the enqueue time. JPMQ stores this data in a table representing the queue. As we have already seen, each message is given an unique MID which is used as a primary key of this table. A foreign key is used to reference the root node of the application data which is stored in the body of a message type.

The database schema used to store application data may be built up from several tables. JPMQ creates a database table for each node type which is contained in the message type. The node type's attributes become columns within these tables and have an appropriate data type assigned. For the references between node types, extra tables are created because they may be (n:m)-relationships at the instance level. Hence, JPMQ creates five tables to store the message type defined in Fig. 2.

As already mentioned, we also need to map the JPMQ queries to the query language SQL. JPMQ uses queries to calculate the PCF and to provide a value-added dequeue operation with user defined ordering and selection criteria. The usage of an RDBMS enables JPMQ to delegate the evaluation of these queries to the storage layer. JPMQ provides a structural object-oriented type system with set-valued references. Hence, SQL as a JPMQ query language is insufficient, because application developers would need to know the mapping of message types to tables, thereby losing the advantages of a more object-oriented type model. Also, the SQL queries resulting from the mapping of message types to tables tend to be very complex. Therefore, we decided to use an OQL-like query language for JPMQ queries, which is aware of set-valued attributes. However, the JPMQ queries have to be mapped to SQL in order to exploit the query power provided by the storage layer.

Explaining the full JPMQ query language is beyond the scope of this paper. Nevertheless, we present some examples which should show that transformation of the queries requires some efforts but is not a critical task. For JPMQ queries we demand some restrictions in order to ease the process of query transformation. For example, the queries

```
select  *
from    mt_ordermessage mt, mt.root o, order_line ol
where   ol in o.order_line
```

and

```
select  *
from    mt_ordermessage mt, mt.root o, o.order_line ol
```

will lead to the same result. Yet, the first one is not a valid JPMQ query because of the free variable 'ol'. The 'from' clauses of a JPMQ query must span an directed acyclic graph, whereas the variables representing message nodes are the nodes and the references are the arcs. Free variables are variables which are not reachable in this graph starting at the root node and following the arcs.

Notice, that the node representing the administrative information of a message is always the anchor of a JPMQ query and is named 'mt_<message type name>'. The attribute 'root' references the root of the message type. Administrative information represented by attributes of node 'mt_<message type name>' can be used in queries too.

The example from Sect. 3.4 follows these restrictions. Executing the selective ordering dequeue operation means transforming OQL statements into SQL statements. First, the selection criterion is evaluated. Then message identifiers of the result set are sorted by the OF and read from the database. At last JPMQ reads the application data regarding to these identifiers. This leads to SQL statements similar to the one below:

```
select  q1.mid as mid, of.sort1 as sort1
from    q_orderqueue q1, (<OF>) as of
where   q1.mid = pcf.mid and q1.mid in (<SC>)
order by sort1
```

In order to evaluate the OF, the term `<OF>` is replaced by an SQL statement:

```
select  m2.mid as mid, count (i1.oid) as sort1
from    q_orderqueue q2, mt_ordermessage_order o1, mt_ordermessage_ref1 ref1,
        mt_ordermessage_orderline ol1, mt_ordermessage_ref2 ref2, mt_ordermessage_item i1
where   q2.root = o1.oid and o1.oid = ref1.foid and ref1.soid = ol1.oid and
        ol1.oid = ref2.foid and ref2.soid = i1.oid
```

Accordingly, the term `<SC>` is replaced by the statement below:

```
select  q3.mid
from    q_orderqueue q3, order o2
where   q3.root = o2.oid and o2.name = 'Smith'
```

The selection and sorting of messages is a two-phase process. In the first phase, the qualifying messages and their order is determined, whereas in the second phase the actual data of the messages is read. In the second phase we take advantage of the MID stored in each record. As described before, JPMQ needs to access every table only once, reading all records which belong to the messages determined in the first phase. JPMQ creates main memory hash tables using the records representing references to ease the transformation of external storage references to Java references. Messages are held in a generic structure within the Java virtual machine.

5.4 Using Java

Our decision for Java([8]) as the implementation platform stems from several features of this programming language and its run-time system. The tight integration of the Java environment into the Internet is just one of the more important features.

With the concepts of RMI (*remote method invocation* [17]) along with *object serialization* as standards for Java, the gaps between JPMQ and applications are easily bridged in a portable manner. In the same line, the JDBC ([16]) standard provides a uniform, low level interface to any RDBMS which must—according to the specification of JDBC—support ANSI SQL-2 Entry Level ([15]). The usage of Java (and RDBMS) relieves us from dealing with heterogeneity aspects, since Java defines the physical layout of all types precisely and the concept of object serialization solves the problems of exchanging data between different platforms. Another feature supporting our decision for Java is the run-time extensibility of applications. In this manner, we can extend the running JPMQ service with the functionality for new message types and appropriate message type services. In the remainder of this section, we introduce the generic data structures used to provide access to messages by application programs and the main functionality of the API.

Data structures. The application receives messages from the queue in the standard generic representation of JPMQ. Instances of the class `JPMQMessage` are used as entry points to the messages. Each node is represented by an instance of `JPMQMessageNode`, which is the superclass of `JPMQMessage`. Instances of `JPMQMessageNode` provide a generic late-binding interface for accessing the node's attributes. Relationships are built from instances of `JPMQReferenceCollection`. This class is a subclass of the Java class `Vector` and offers functionality for direct access to elements using an index as well as iterators to enumerate the records included. The generic classes come with the following operations:

```
JPMQValueType            JPMQMessageNode.getValue(String attrName);
JPMQReferenceCollection  JPMQMessageNode.getNodes(String attrName);
String                   JPMQMessageNode.getNodeName();
JPMQMessageNode          JPMQMessage.getRootNode();
String                   JPMQMessage.getTypeName();
Enumeration              JPMQReferenceCollection.getElements();
```

As application programmers should not be burdened with the internal mechanisms of the generic representation, an external representation for each message type may be generated using the message type definition. This external representation will act as a wrapper for the generic one in the sense that it presents a regular (Java) type to the application, but is a subclass of the generic type and provides functionality to switch between external and internal representation. In our example, five classes (`MTOrder` (subclass of `JPMQMessage`), `MOrder`, `MOrderline`, `MItem` (subclasses of `JPMQMessageNode`), `MOrderlineCollection`, `MItemCollection` (subclasses of `JPMQReferenceCollection`)) would have been generated. Let us examine `MTOrder` to give an idea of how the generated classes may be used. The interface will include methods to access the value-typed attributes as well as the set-valued attributes, as for example:

```
String                MOrder.getName();
void                  MOrder.setName(String name);
MOrderlineCollection  MOrder.getOrderline();
```

6 Conclusions and Outlook

In this paper we have introduced JPMQ which may be used as a base service to support reliable exchange of data between applications even through the Internet.

Middleware services must be easy to use and to support a programming model which fits into the application development environments. In the field of MOM, this means both supporting a type model matching the object-oriented programming paradigm common in today's software projects and being applicable in a wide range of application domains. During the development of JPMQ these requirements have influenced our decision to support an application-independent type model for the exchange of structured data between applications.

JPMQ's structured type model was presented as the base of the advanced functionality provided by our MOM service. PCFs have been introduced, enabling the QS to schedule messages based on their contents. In addition, the dequeue operation was enhanced by introducing ordering and selection criteria. While selection criteria enable JPMQ-developed applications to decide which messages are needed at run-time, the ordering criteria makes it possible to change the order in which messages are received. Both are helpful extensions of the dequeue operation without loosing the simplicity of MOM services.

Our MOM service was designed to support reliable exchange of data. The ACID properties of database transactions were compared to the idea of queue transactions in order to detect similarities. Hence, JPMQ provides atomic queue transactions, even in the presence of failures. A logging scheme was introduced which enables JPMQ to recover a transaction-consistent state after a system crash. In addition, we argued the need of an isolation protocol to use JPMQ in multi-user environments with several application programs accessing one queue concurrently (as producers and/or consumers). Different isolation protocols were introduced in order to support a set of useful qualities of service.

We have also shown the advantages of using relational database technology. The properties of RDBMSs, i. e., ACID transactions and query interface, are useful with respect to the implementation of the service-specific operations and the transaction semantics of JPMQ. By using an RDBMS, message persistence is easy to implement. The ACID properties of database transactions enable us to achieve the service-specific isolation and logging protocols relying on an operation-consistent state regarding the data, the isolation state, and the log entries of JPMQ. Additionally, the set-oriented operations of an RDBMS are helpful to implement the isolation and logging protocols. Furthermore, we have taken advantage of the expressiveness of SQL to evaluate the PCFs as well as the ordering and selection criteria.

Java has been designed to ease development of distributed object systems (DOS) in the domain of Internet applications. JPMQ has been implemented using this technology, and we have shown that the provided concepts are really useful. Unfortunately, Java applications are still less efficient than, e. g., applications developed in C++. In order to allow the service to be transparently replaced by a more efficient implementation, the next step will be to enable access to JPMQ via CORBA interfaces. This will allow both the use of JPMQ by applications developed with other programming languages and the provision of a new service based on a different technology. We are very interested in the problems related to the use of a much less object-oriented technology, e. g., the C++ programming language.

Another aspect of future work is to achieve more failure tolerance in distributed object systems. Until now, a JPMQ node is identical to a JPMQ service. If the node fails, the service also fails. Hence, our goal is to build a JPMQ node by a group of JPMQ services. In case of a failure of one service, another service should be able to take over work.

Acknowledgments

We would like to thank Th. Härder and N. Ritter for their helpful comments on an earlier version of this paper.

References

1. B. Blakeley, H. Harris, R. Lewis: Messaging and Queuing Using MQI. McGraw-Hill, Inc. (1995)
2. P. A. Bernstein, M. Hsu, B. Mann: Implementing Recoverable Requests Using Queues. Proc. ACM SIGMOD (1990)
3. G. Booch: Object-Oriented Analysis and Design with Applications. Benjamin/Cummings (1994)
4. P. A. Bernstein, E. Newcomer: Principles of Transaction Processing. Morgan Kaufmann Publishers, Inc. (1997)
5. R. G. G. Cattel, edt.: The Object Database Standard: ODMG-93. Morgan Kaufmann Publishers, Inc. (1996)
6. G. Coulouris, J. Dollimore, T. Kinderberg: Distributed Systems: Concepts and Design. Addison-Wesley Publishers Ltd. (1994)
7. Sybase, Inc.: dbQ User Guide & dbQ ReferenceGuide.
   Available at http://www.sybase.com/products/internet/dbq/ (June 1997)
8. J. Gosling, B. Joy, G. Steele: The Java Language Specification. Addison-Wesley, Inc. (1996)
9. J. Gray, A. Reuter: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, Inc. (1993)
10. T. Härder, A. Reuter: Principles of Transaction Oriented Database Recovery. ACM Computing Surveys Vol. 15, No. 4 (1983)
11. I. Jacobson, M. Ericsson, A. Jacobson: The Object Advantage. ACP Press (1995)
12. M. Jotzo: JPQMS: Ein verteilter, objektorientierter Kommunikationsdienst für asynchrone Transaktionsverarbeitung. Diploma-Thesis (in German), Universität Kaiserslautern (December 1997)
13. J. Melton, A. R. Simon: Understanding the new SQL: a complete guide. Morgan Kaufmann Publishers, Inc. (1993)
14. R. Orfali, D. Harkey, J. Edward: The Essential Client/Server Survival Guide. John Wiley & Sons, Inc. (1996)
15. American National Standards Institute (ANSI): Database Language SQL. Document ANSI X3.135-192
16. Sun Microsystems, Inc.: JDBC: A Java SQL API. Available at ftp://ftp.javasoft.com/pub/jdbc/jdbc-spec-0120.ps (March 1998)
17. Sun Microsystems, Inc.: RMI—Remote Methode Invocation. Available at http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/ (March 1998)
18. H.-P. Steiert, J. Zimmermann: JPMQ—An Advanced Persistent Message Queuing Service (internal report).
    Available at http://www.uni-kl.de/AG-Haerder/publications/p1998.html