# Market Requirements Document

## Feature Name: Distributed Queue Manager

**Version:** 1 **Completed By:** Leon Guzenda

**Date Submitted:** 06/19/07

### Description of the Problem

#### *Background*

Many applications need to place objects on a persistent queue for subsequent processing. The objects may represent actions or be data that has been selected for subsequent processing by the current or a separate algorithm.

#### *Problem*

Although there are many ways to create a queue of objects in Objectivity/DB, such as a persistent linked list, the current mechanisms do not perform well enough when there are large numbers of objects arriving from multiple sources. This is because container level locking only allows one writer at a time. In most cases, both the placement of objects on the queue and removing them from the queue requires an update lock.

### Description of the Requested Feature

#### *Functionality*

1. A mechanism that allows multiple writers to add objects to a logical First In First Out (FIFO) queue without being blocked by other writers.
2. An iterator mechanism that can remove objects from the FIFO queue with minimal impact on other writers and queue readers.
3. The logical queue must automatically expand across container and database boundaries in the event that entries on the queue are not being removed fast enough to prevent file, container or database limitations being reached.

#### *Performance*

1. When multiple writers are adding entries to the logical queue the perceived performance at each node must be as close as possible to the speed at which a single writer can add entries.

2. When multiple readers are servicing a single logical queue the time to iterate over the queue should be within one percent of the time it would take a single reader to iterate over a queue in a single container.

**Part of an existing feature or does it require another feature, if so, which one?**

- This will be an optional feature of Objectivity/DB.

**How is this problem being solved now, and why isn't that acceptable?**

Customers are building their own solutions, often having to write queue servers or use third party message queuing alternatives.

**What languages must support this capability?**

- All APIs, starting with C++, Java and .Net for C#.

**Which platforms must be supported?**

- All platforms.

**Do any competitors already have this feature?**

- GemFire (GemStone)
- JADE

**Customers who require this feature**

Many customers, particularly in the "Monitoring, Analysis and response" category have built their own queuing mechanisms or used middleware. Typical users would include:

- Engineering/Manufacturing
  - Data Acquisition & Telemetry
  - Process Control & Automation
- Complex Financial
- Government
  - Control, Communications, Computers. Intelligence, Surveillance, Target Acquisition and Reconnaissance (C4ISTAR)
  - Data Fusion
  - Intelligence Community
- Health Sciences and Medical Equipment
- Scientific Computing

- Internet Related
  - eCommerce
  - Internet Infrastructure
  - Software as a Service
- Complex IT, especially Complex Event Stream Processing

- Telecom

  - Advanced Intelligent Network
  - Element Management System
  - Network Management
  - Network Operations Center
  - Operations Support Systems
  - Personal Communications Services

**Revenue at risk, or which could be won**

- An efficient distributed, shared queuing mechanism should increase our chances of winning business in the following market categories:
  - **Design and Simulation**
  - **Monitoring, Analysis and Response Systems**

**When is this required?**

- Post Release 10.

**Additional Notes**

1. The event notification capabilities described in a separate MRD (Unified Date and Time) could leverage a Queue Manager if it is implemented in time.

2. We will also need:

   - Marketing collateral, including promotional material and a special area on our web site.
   - Technical Publications.
   - New QA material to prove that the API works and is interoperable with other platform and language combinations.

3. Licensing costs are to be determined.

4. Appendix A describes a mechanism that should be considered as a potential solution to this requirement.

## APPENDIX A – A Mechanism For Supporting A Distributed, Shared Queue

### Introduction

One simple way to implement a shared queue would be to build a server that receives messages containing the object identifiers (OIDs) of objects to be put on the queue. Multiple writers could send messages to it and multiple readers could send read or "read and remove" requests. An Advanced Multithreaded Server (AMS) with a modified Objectivity Open File System (OOFS) layer could implement a suitable caching strategy, such as "Retain in memory until requested."

While simple to implement, this mechanism could rapidly become a performance and scalability bottleneck. It would also be a single point of failure. So, we need a mechanism that is distributed, fast and failsafe.

### Ordering

"**FIFO** is an acronym for **First In, First Out**. This expression describes the principle of a queue or first-come, first-served (FCFS) behavior: what comes in first is handled first, what comes in next waits until the first is finished, etc. Thus it is analogous to the behavior of persons queuing (or "standing in line", in common American parlance), where the persons leave the queue in the order they arrive." – Wikipedia 06/19/07.

The ordering (of objects or OIDs) may be achieved in many ways, including:

a) Sequential physical placement in memory or a file.
b) A forwardly linked list (or degenerate use of other ordered collections).
c) A vector or indexed array.
d) A chronologically maintained association (uni-directional or bi-directional) between a queue object and queue members.
e) A timestamp.

Mechanisms a) through d) will only work efficiently with Objectivity/DB if the queue is maintained within a single container. Mechanism e) can work in a fully distributed environment, but it has two significant drawbacks:

- All writers must use the same clock.
- Finding successive queue entries without an index (which is itself a concurrency and performance problem) is expensive.

### Proposed Algorithm

1. Each writer uses a dedicated Queue Fragment container that represents the writer's contribution to the distributed, shared queue. These containers are registered in a

single Distributed Queue Management container that represents the distributed, shared queue.

2. The writer can place one or more objects or OIDs into an object that is an instance of ooSharedQueueEntry (or a subclass). The ooSharedQueueEntry has an ooFederation_Timestamp field and an optional VArray of OIDs, each one being a queue entry.
3. The timestamp to be inserted into the ooFederation_Timestamp field is calculated from a federation timestamp obtained from the lock server at the start of each transaction. This timestamp is compared with the local system clock and the offset is stored for the duration of the transaction. The value stored in the ooFederation_Timestamp is the local system time plus the offset. All of the times will be held in the format described in the Objectivity Timestamp MRD. The mechanism for dealing with multiple lock servers is described later.
4. The iterator that returns and generally removes entries from the queue can run in a client or be implemented with the aid of a Queue Server process.
5. The iterator looks in the Distributed Queue Management Container to determine the number and identities of the Queue Fragment containers to be searched. It passes this information to a Parallel Query Engine task splitter that subdivides the work across PQE Query Agents.
6. Each Query Agent scans its target Queue Fragment container and uses a local sort to return queue entries and their timestamps in chronological order. The sort can be avoided if the chronological ordering can be guaranteed by some other mechanism.
7. The client side iterator then marshals the returning streams into chronological order before returning results to the caller.

One potential problem is that if the queue servicers are not running continuously, or keeping up with a growing queue, there may be large gaps in time between the entries stored in different Queue Fragment containers. This problem can be reduced by having the Query Agent remember (transiently or persistently) the earliest and latest timestamp in a container. This information could be returned in a setup query directed at all of the containers, followed by a simple scheduling algorithm that would help minimize the gaps between bunches of returned entries. A Query Agent would normally only return a range of queue entries once. The client process would then delete the queue entries, unless handling them is delegated to a separate server.

If there are multiple lock servers they should use the federated database to store and retrieve a master time obtained from any one of them. Alternatively, we could limit the range of a distributed queue to have its Queue Fragments stored in databases under the control of a single partition’s lock server. The queue entries could still reference objects anywhere in the federation.

New entries could be arriving while the iterator is receiving and marshalling entries. Having the iterator supply the current timestamp to the Query Agents so that they do not return any entries created after that time can solve this problem.

The polling nature of the algorithm could be made more efficient if there is an effective event notification mechanism that can be used to place clues in the Distributed Queue Management Container or to trigger the processes servicing the queue.