IONA

Technologies

Making Software Work Together™

OrbixOTS 3.0 Performance White Paper

IONA Technologies
January 2000

IONA

Technologies

Making Software Work Together™

M2382

IONA Technologies

Making Software Work Together™

Summary

Applications written using OrbixOTS (the implementation of the OMG CORBA Object Transaction Service from IONA Technologies) benefit from the important transactional ACID properties of atomicity, consistency, isolation, and durability. However, these benefits come at the cost of reduced performance. This guide explains why these costs arise and discusses the options available when using OrbixOTS to achieve maximum performance.

IONA Technologies

Making Software Work Together™

IONA Technologies

Making Software Work Together™

Table of Contents

Making Software Work Together™

Introduction

OrbixOTS is IONA Technologies' implementation of the OMG CORBA Object Transaction Service. It provides application developers with the well-known benefits of the ACID transactional properties of atomicity, consistency, isolation, and durability. However, these benefits come at the cost of reduced performance. This guide explains why these costs arise and discusses your options for achieving maximum performance. These options relate to application design, and system environment and configuration issues. The focus is mainly on increasing throughput (that is, the number of transactions per second). Some response time issues are also covered.

In general there are four areas where the use of transactions affect application performance. These areas are:

1. Recovery logging.

2. Two-phase commit messages.

3. Propagation context.

4. Data isolation.

This white paper examines each of these areas in turn. Some miscellaneous areas are also examined, along with related configuration variables. This white paper concludes with a quick summary of some key performance tips.

This document assumes knowledge of CORBA, the CORBA Object Transaction Service, the CORBA Concurrency Control Service, the X/OPEN XA specification, and OrbixOTS.

Recovery Logging

The durability property of transactions is achieved by saving information to stable storage at key points in the lifetime of a transaction. This is necessary so that the coordinator and all participants maintain a consistent view of the transaction in spite of any failures that may occur. For example, in the two-phase commit protocol, once the coordinator has made the decision to commit, a record of this fact must be forced to stable storage before the protocol can continue. If the process or the machine hosting the coordinator crashes before all participants are notified of the commit decision, the messages can be replayed during recovery by examining this record.

Writing to stable storage can be orders of magnitude slower than normal processing, making the durability property one of the most expensive parts of a transaction. All records do not need to be forced to stable storage, however. For example, a participant's response to a commit message does not need to be forced—a failure will only result in a redundant replaying of the commit message.

OrbixOTS implements durability using a transaction log. Logically the transaction log can be viewed as an append-only storage device that stores transaction records sequentially. In reality it is implemented as a finite size file (typically 8 megabytes) in which unused records are removed to prevent the log from filling up.

The transaction log can be composed of several files or partitions. It can also be mirrored to prevent loss of information due to media failures. For a mirrored log, the data records are written twice: once to the normal log, and once to the mirror log. In OrbixOTS it is also possible for one server to use the transaction log of another OrbixOTS server.

From a performance point of view, a number of steps can be taken to reduce the overheads of logging:

- Always use raw disk partitions rather than normal files for log files.
- Take advantage of the group commit feature to increase disk I/O throughput.
- Reduce the amount of data logged by careful application design.
- Do not mirror the transaction log.
- Do not use the remote logging feature.

Raw Disk Partitions

On systems where they are supported (for example, Solaris and HP-UX), using raw disk partitions instead of normal files for the transaction log results in a dramatic increase in performance. By using a raw disk partition, OrbixOTS bypasses the operating system's file system structures and associated overheads. In some situations, simply switching to using a raw disk partition can triple the transaction throughput.

Using raw disk partitions is no different from using normal files. Take care that the raw version of the partition is used and not the buffered version, and that the partition is not allocated to a file system. For example, the following code shows how an OrbixOTS server is initialized to use the raw disk partition `dev/rdsk/c0t10d0s2`:

```
// C++ (OrbixOTS servers only)
// Error handling omitted for clarity.

int
main()
{
// …
OrbixOTS::Server_var ots = OrbixOTS::Server::IT_create();

ots->logDevice("/dev/rdsk/c0t10d0s2");
ots->restartFile("/local1/tlog.restart");
ots->mirrorRestartFile("/local2/tlog.mirror");
// …
ots->init();
}
```

On systems where raw disk partitions are not available or practical, it is important that the transaction log resides on a fast local disk.

Group Commit Feature

In a busy transactional system the transaction log can quickly become a performance bottleneck. As each transaction commits, it must wait for all currently committing transactions to complete their disk writes before it can proceed. Because disk writes take so long relative to other activities, it is better to group several disk writes together. The *group commit* feature of OrbixOTS can dramatically increase the transaction throughput.

As an analogy, consider the situation where a number of people are out for a meal together in a restaurant. If the waiter takes each person's order separately, and the chef cooks the meals sequentially, the person whose order was taken first will be served quickly, but the second customer will be waiting twice as long, the third customer will be waiting three times as long, and so on. On the other hand, if the waiter takes an order from everyone at the table at the same time, and the chef is able to prepare the meals together, everyone will receive their food at about the same time. The first customer may have to wait a little longer but the time taken to serve the remaining customers is dramatically reduced.

The group commit feature is automatically available in OrbixOTS. Application programmers must, however, ensure that their servers use concurrent transactions by using threads as much as possible and specifying a value of concurrent to the `OrbixOTS::Server::impl_is_ready()` operation. The latter permits concurrent requests from different transactions by using a thread pool for user requests.

The user request thread pool uses a high watermark/low watermark scheme to control the maximum and minimum number of threads. For more information see "User Request Thread Pool" on page 16.

By designing your applications to exploit the group commit feature in OrbixOTS several times, you can increase the transaction throughput. This is very useful when you are forced to use slow disks or when normal files have to be used instead of raw disk partitions.

Reducing the Amount of Data Logged

One obvious way of speeding up disk writes is to reduce the amount of data that needs to be written. By careful application design you can decrease the amount of data that OrbixOTS writes to stable storage with each transaction. One or more of the following is recommended:

- Reduce the number of servers involved in a transaction.
- Reduce the number of XA resource managers registered with servers.
- Reduce the number of resource objects registered with transactions.

For example, if your application implements recoverable objects by using the `CosTransactions::Resource` interface, it is better from a logging point of view to register a single resource object for all modified recoverable objects, rather than using one resource object for each recoverable object.

Mirroring Logs

When the transaction log is mirrored (using the `otsadmin` tool), each write to the log is performed serially to each of the mirrors. This ensures that at least one of the logs contains the correct information. The time taken for each write is, however, proportional to the number of mirrors used. Therefore, to increase both response time and throughput, the number of mirrors should be minimized. In most situations one mirror is enough.

Use of Logging Servers

It is possible to minimize the number of transaction logs in a system by making one or more OrbixOTS C++ server use another OrbixOTS C++ server's log, by using the `OrbixOTS::Server::logServer()` operation. For example, the following code initializes a server to use the transaction log of the `otstf` transaction factory server.

```
// C++ (OrbixOTS servers only)
// Error handling omitted for clarity.

int
main()
{
OrbixOTS::Server_var ots = OrbixOTS::Server::IT_create();
ots->logServer("OrbixOTS_TransactionFactory");
//…
ots->init();
// ..
}
```

Although this feature can be useful, it increases the already high cost of logging transactional records.

Two-Phase Commit Messages

The *atomicity* property of distributed transactions is achieved using the standard two-phase commit protocol. This protocol is designed so that all participants in a distributed transaction agree on the final outcome of the transaction, even in the event of failures affecting the participants, the coordinator, and the communication paths.

In the two-phase commit protocol, a "prepare" message is first sent to each participant. Each participant votes on whether to commit or rollback the transaction. The coordinator gathers all votes and makes a final irrevocable decision to commit or rollback the transaction. Therefore, a transaction with *n* participants typically requires 2 X *n* messages to successfully commit. This means that the two-phase commit protocol, along with logging for recovery, is one of the most expensive parts of transaction management.

The following steps can be taken to reduce the costs associated with the two-phase commit protocol:

- Reduce the number of transaction participants.
- Make use of the read-only vote to the prepare request.
- Use the one-phase commit XA optimization.
- Use dynamic XA resource manager registration.

Reduce the Number of Transaction Participants

Because the number of round-trip messages sent during the two-phase commit protocol is proportional to the number of participants in a transaction, reducing the number of participants will reduce the overhead of committing that transaction. This is particularly relevant if a participant is separated from the coordinator by a slow communications line.

In OrbixOTS a participant can be one of the following:

- A registered XA resource manager.
- A registered resource object. This can be either due to an application or an interposed transaction to support interoperability between different OTS implementations.
- Another recoverable OrbixOTS server visited during the transaction.

If you are designing an application, the cost of the two-phase commit protocol can be minimized by reducing the scope of transactions to involve only a few servers (preferably only one), and reducing the number of resource objects registered with transactions.

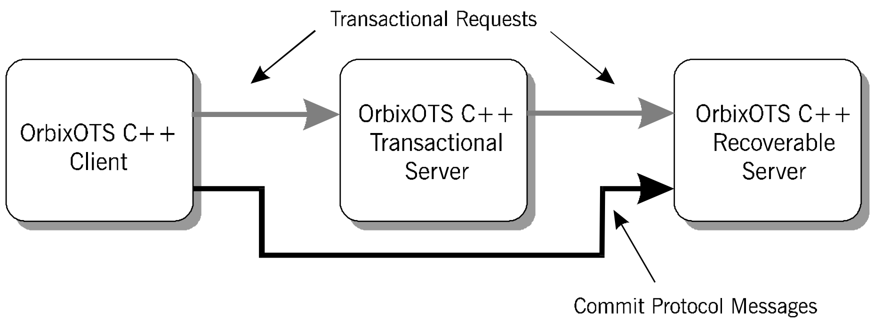

Figure 1—Optimized Commit Protocol

OrbixOTS can be optimized so that only relevant servers are involved in the commit protocol. For example, consider the scenario shown in Figure 1, where a client makes transactional invocations on an OrbixOTS server, which in turn makes a transactional invocation on a recoverable OrbixOTS server. The first server is not recoverable because it does not have any registered XA resource managers and does not register any resource objects with transactions. As a result, when the client commits the transaction, the first server is not involved in the commit protocol and OrbixOTS forwards the commit request to the second recoverable server. In this scenario, the second server becomes the transaction coordinator.

Read-Only Participants

If a participant does not modify or change the data, it does not need to be fully involved in the two-phase commit protocol. For example, if an object supporting the `CosTransactions::Resource` interface returns `VoteReadOnly` from the `prepare()` operation, the object is no longer involved in the transaction, and the coordinator does not need to send a final commit or rollback message.

Dynamic XA Registration

When an XA resource manager is registered with an OrbixOTS server, it is normally included in all transactional activities. This is known as *static* *registration*, and means that the resource manager is always involved in the two-phase commit protocol, regardless of whether any data belonging to the resource manager is accessed during a transaction. Statically registered resource managers also have their `xa_start()` and `xa_end()` functions called at the start and end of transactional work, respectively.

OrbixOTS supports another form of XA registration called *dynamic registration*. Here, the resource manager decides when it wants to become involved in a transaction, if at all. If it wants to be involved in a transaction, it calls OrbixOTS asking to become a participant. Otherwise, the resource manager is not involved in the transaction and no prepare message is sent at commit time.

Dynamic XA registration is set using the flags in the `xa_switch_t` structure provided by the resource manager's XA library. Refer to your resource manager's documentation for more information on this option.

One-Phase Commit XA Optimization

The X/Open XA specification includes an optional one-phase commit protocol that can be used where there is only one active resource manager involved in a transaction. This avoids using the two-phase commit protocol and reduces the amount of logging. This optimization feature is available in OrbixOTS C++ servers, under certain conditions, through the `tmxa_SetUsesOnlyLocalXaWork()` Encina Toolkit function. To turn on this feature, use the following code after initializing OrbixOTS:

```
// C++ (OrbixOTS server only)
// Turn on one-phase commit/XA optimization.
tmxa_status_t status;
status = tmxa_SetUsesOnlyLocalXaWork(0,
TMXA_NEW_TOP_LEVEL_TIDS,
TMXA_ONLY_LOCAL_XA_WORK);
if (status != TMXA_SUCCESS)
{
// Error handling
}
```

Note that the one-phase commit/XA optimization can only be used if the following two conditions are met:

1. All XA resource managers are registered with a single OrbixOTS server.

2. No `CosTransactions::Resource` objects are registered with transactions.

Propagation Context

When an invocation is made on a transactional object (that is, an object that supports the `CosTrasactions::TransactionalObject` interface), additional information is added to the request containing information about the transaction. To support interoperability between different OTS implementations, a standard `CosTransactions::PropagationContext` structure is used (see Figure 2), and the data is placed in a standard General Inter-ORB Protocol (GIOP) service context.

```
// IDL (in CosTransactions module)

struct otid_t
{
long formatID;
long bqual_length;
sequence <octet> tid;
};

struct TransIdentity
{
Coordinator coord;
Terminator term;
otid_t otid;
};

struct PropagationContext
{
unsigned long timeout;
TransIdentity current;
sequence <TransIdentity> parents;
any implementation_specific_data;
};
```

Figure 2—Transaction Propagation Context

The cost of encoding and decoding the propagation context, and adding it to each transactional invocation, can have an impact on transactional performance.

Reducing Context Size

The size of a propagation context is most affected by the number of nested transactions and the size of the `implementation_specific_data` field. Nested transactions have an obvious affect because the propagation context includes the identifier for all of the transaction's parents (that is, the `sequence<TransIdentity> parents` field). Therefore, reducing the number of nested transactions that are *propagated* directly, reduces the size of the propagation context and the complexity of encoding and decoding the propagation context.

The `implementation_specific_data` field is more complex because it contains low level information used by the Encina Toolkit. For transactions created by OrbixOTS Java applications, this field is always empty. However, when OrbixOTS C++ applications interact, the size of the field is affected by the following:

- The number of servers visited by the transaction.
- The number of `CosTransactions::Resource` objects registered with the transaction.

Note that when OrbixOTS C++ applications interact, the full propagation context is not used. Instead only the data contained in the `implementation_specific_data` field is exchanged using a proprietary GIOP service context. This reduces the amount of data exchanged, and reduces the time spent encoding and decoding the context. This optimization feature can be disabled by setting the OrbixOTS configuration variable `OTS_NO_OPTIMIZE_PROPAGATION` to `TRUE`. Interoperability with foreign OTS implementations is not affected by this optimization feature.

Reducing Use of the Propagation Context

Designing your OTS applications to limit the number of times a transaction must be propagated can increase performance. Limiting the number of servers visited by the transaction and delaying the creation of transactions as late as possible in the propagation achieves this. For example, if a client is only performing one server operation per transaction, the transaction can be created in the server to prevent unnecessary propagation.

Alternatively you can take advantage of the *transaction policy* feature in OrbixOTS. Normally, an object is considered transactional if the IDL interface it supports inherits from the `CosTransactions::TransactionalObject` interface. Invoking on such an object always requires the sending of a propagation context. However, in OrbixOTS you can specify that an object *allows* a transaction. This means that a propagation context is sent only if there is a current transaction.

As an example of using the *allows* policy, consider the following client code that first makes a lodgement to a bank account and then transfers data between two accounts:

```
// C++ (OrbixOTS clients and servers)
// Error handling omitted for clarity.

TransAccount_var acc1 = ...
TransAccount_var acc2 = ...

// Make a lodgement (no client transaction required).
acc1->makeLodgement(100.00);

// ...

// Transfer funds between two accounts (this requires a
// transaction).
CosTransactions::Current_var current = ...

current->begin();
acc1->makeLodgement(150.00);
acc2->makeWithdrawal(150.00);
current->commit(1);
```

Because the first lodgment only involves one operation, no transaction needs to be created. However, the subsequent transfer involves two operations—a lodgment and a withdrawal—and a transaction is required to ensure that ACID properties are guaranteed for both accounts. This transaction is necessary even if both accounts use the same database and the same server.

When using the *allows* policy, each server operation must check for the existence of a transaction. If none is propagated, the operation must create a new transaction in which the operation executes. For example, the `TransAccount::makeLodgement()` operation might be coded as follows:

```
// C++ (OrbixOTS servers only)
// Error handling omitted for clarity.

void
TransAccountImpl::makeLodgement(
CORBA::Float amount,
CORBA::Environment&
)
```

```
{
// Check to see if there is a current transaction.
//
CosTransactions::Current_var current = ...;
boolean local_tran = false;
if (current->get_status() ==
CosTransactions::StatusNoTransaction)
{
// No transaction was propagated so create one.
local_tran = true;
current->begin();
}

// Do the lodgement...

// If no transaction was propagated the locally created
transaction
// must be committed.
if (local_tran)
{
current->commit(1);
}
}
```

Setting the *allows* policy on an object must be done in both the client and the server. There are three ways to do this:

1. Use `OrbixOTS::setObjectTransactionPolicy()` for the object reference.

2. Use `OrbixOTS::setInterfaceTransactionPolicy()` for the object interface. This way all objects supporting the interface use the allows policy.

3. Use `OrbixOTS::setDefaultTransactionPolicy()` so that all objects have the allows policy by default.

For example, to set the allows policy for all objects supporting the `TransAccount` interface the following code can be used:

```
// C++ (OrbixOTS clients and servers)

OrbixOTS::setInterfaceTransactionPolicy(
“TransAccount”,
OrbixOTS::transactionAllowed);
```

Note also that the OTS 1.1 specification does not make it mandatory for a propagation context to be sent in reply messages. OrbixOTS always returns a context for transactional invocations between two OrbixOTS C++ applications, but in other situations, such as for a Java client, a context is not returned unless the OrbixOTS configuration variable `OTS_ALWAYS_RETURN_CONTEXT` is set to `TRUE`.

Data Isolation

The *isolation* property of transactions ensures that changes made to shared data by a transaction are not made visible to other transactions until the transaction commits. The data access layer of the application cooperates with OrbixOTS to provide the isolation property. For example, if your application uses an XA database, the database typically supports the isolation property using data locks.

Supporting isolation can reduce the amount of concurrency in an application, particularly when many transactions compete for the same set of shared data. To reduce the potential for conflicts, applications should be designed so that transactions are of short duration. Each XA resource manager may have options to further increase performance; refer to your resource manager's documentation for more details.

For pure CORBA OTS applications using `CosTransactions::Resource` objects, the best way of achieving data isolation is by using the lock-based Object Concurrency Control Service (OCCS) that is supplied as part of OrbixOTS.

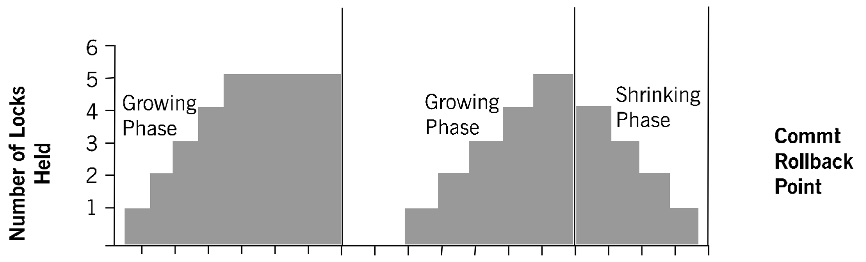

Figure 3—Strict and Non-Strict Two-Phase Locking

When using the OCCS, it is important that the *two-phase locking* protocol is used. This means that once an application starts to release locks, no further locks can be acquired. With two-phase locking, a transaction has a growing phase in which locks are acquired, and a shrinking phase in which locks are released.

Normally, *strict* two-phase locking is used where all locks are released together at the end of a transaction. This is supported in the OCCS using the `drop_locks()` operation. Strict two-phase locking provides the maximum degree of isolation, but can restrict concurrency. An alternative strategy is *non-strict* two-phase locking in which locks can be released earlier (see Figure 3). This option is supported in the OCCS with the `release_lock()` operation.

If it is appropriate for your application, non-strict two-phase locking should be used. However, releasing locks before a transaction has committed, makes changes visible to other transactions. If the first transaction is rolled back, these transactions must also be rolled back.

Miscellaneous

This section describes how to tune the user request thread pool and also examines some miscellaneous performance areas not covered in previous sections.

User Request Thread Pool

Each OrbixOTS C++ server provides a user request thread pool. This is most useful when the server's serialization mode is concurrent, because it allows both concurrent transactions and concurrent requests.

The user request thread pool uses low and high watermarks to control the number of active threads. Initially there is a single thread in the pool. As concurrent requests enter the server, the number of threads increases up to a maximum of the high watermark. Beyond this point the requests are queued until a thread becomes free. As the number of concurrent requests decreases, the number of threads in the pool also decreases to a minimum of the low watermark. Figure 4 shows the number of threads and concurrent requests for a thread pool with a low watermark of `5` and a high watermark of `10`.

Making Software Work Together™

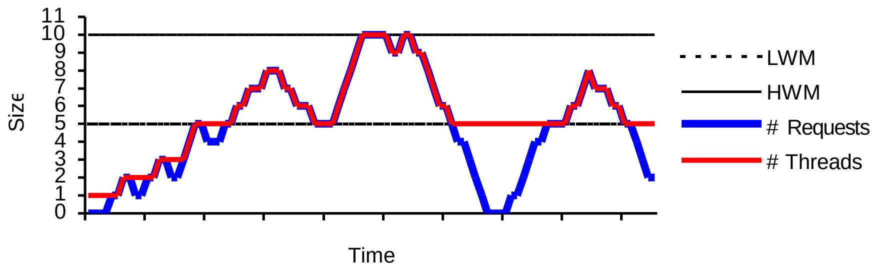

Figure 4—User Request Thread Pool

The low and high watermarks are controlled by the two OrbixOTS configuration variables `OTS_TPOOL_LWM` and `OTS_TPOOL_HWM` respectively. When setting these values, the low watermark should be set to the expected average load, and the high watermark should be set to allow for the expected peak number of concurrent requests. The default values are `5` and `10`. Once the high watermark is reached, requests are queued and this can lead to deadlock depending on the nature of the application.

Normally when a user request thread is scheduled, it is dispatched immediately. In certain circumstances, however, it is desirable to wait until other active requests complete. For example, if the new request on behalf of a transaction is competing for data locks with other requests working for other transactions, waiting for the other requests to complete can prevent the new request from blocking. This feature can be turned on by setting the configuration variable `OTS_DISPATCH_YIELD` to `TRUE` (the default is `FALSE`).

There are also separate request thread pools for admin, logging and transaction protocol messages. Each of these pools has a low watermark of `5` and a high watermark of `50` that cannot be changed. The transaction protocol pool can be disabled by setting the OrbixOTS configuration variable `OTS_OOB_SYNCHRONOUS` to `TRUE`; see “Out-of-Band Messages” on page 19 for more details on this configuration variable.

Caching Resource Manager Data

If you are using OrbixOTS in conjunction with an XA-compliant database, it is often a good idea to provide a per-transaction cache in your application servers. Depending on the nature of the application, this can reduce the number of times the database is accessed in the course of a transaction. Caching with XA resource managers is supported in OrbixOTS through the use of the `CosTransactions::Synchronization` interface:

```
// IDL (in CosTransactions module)

interface Synchronization : TransactionalObject
{
void
before_completion();

void
after_completion(
in Status s
);
};
```

To implement a per-transaction cache, an instance of an object supporting the `Synchronization` interface is registered with the transaction. When the transaction is committed, the `before_completion()` operation is invoked before the OTS attempts to interact with the XA resource manager to commit the modifications. The `before_completion()` operation only needs to flush any modified entries in the cache to the resource manager. The `after_completion()` operation is invoked after the transaction has committed or rolled-back and the final status of the transaction is passed as an argument to the operation.

Do Not Wait for Heuristics

When committing a transaction you have the choice of whether to wait for heuristic outcomes or to return as soon as the coordinator has made its decision on the outcome of the transaction. If your application does not support heuristic outcomes, the response time can be increased by always ignoring them when committing transactions:

```
// C++ (OrbixOTS clients and servers)
// Error handling omitted for clarity.

CosTransactions::Current_var current = ...
current->begin();
// Do transactional work...

// Commit without waiting for heuristics.
current->commit(0);
```

Use of the Current Pseudo Object

The `CosTransactions::Current` interface provides a means of associating transactions with the current thread of control. The OrbixOTS implementation of this interface provides efficient management of transactions by going directly to the Encina Toolkit layer. This means that the CORBA objects representing the `Control`, `Coordinator`, and `Terminator` interfaces are not created until they are required. If possible you should avoid calling the operations `Current::get_control()` and `Current::get_transaction_name()` to avoid the creation of the CORBA objects.

Out-of-Band Messages

OrbixOTS C++ clients and servers exchange messages as part of the commit, rollback and recovery protocols. These messages are exchanged outside the normal client-server application interaction and are termed *out-of-band* (OOB) messages. Normally, each OOB message is first placed in a queue on both the sending and receiving side before being processed by another thread. In environments with a high transaction volume this increases the scalability and performance of OTS applications. However, in certain environments it might be better to avoid queuing these OOB messages. This can be done by setting the OrbixOTS configuration variable `OTS_OOB_SYNCHRONOUS` to `TRUE`.

Configuration Variables

Summary of OrbixOTS configuration variables related to performance. All variables listed below are in the `OrbixOTS` scope.

Configuration Variable Name	Description	Default Value
`OTS_TPOOL_LWM`	The low watermark for the user request thread pool. Once the low watermark is reached, the number of threads in the pool never falls below this value. Only relevant for OrbixOTS C++ servers.	`5`
`OTS_TPOOL_HWM`	The high watermark for the user request thread pool. This represents the maximum number of concurrent user requests that can be active in a server at any one time (new requests are queued until a thread becomes free).	`10 x OTS_TPOOL_LWM`

	Only relevant for OrbixOTS C++ servers.	
OTS_DISPATCH_YIELD	If set to TRUE, threads servicing user requests yield just before being dispatched; otherwise they are dispatched immediately. Only relevant for OrbixOTS C++ clients and servers.	FALSE
OTS_OOB_SYNCHRONOUS	If set to TRUE, out-of-band messages (for example, two-phase commit messages) are sent immediately instead of being queued. Only relevant for OrbixOTS C++ clients and servers.	FALSE
OTS_INTEROP	If set to TRUE, CORBA-compliant propagation contexts are sent in the standard GIOP service context; otherwise the context is piggy-backed on requests using Orbix filters.	FALSE
OTS_ALWAYS_RETURN_CONTEXT	If set to TRUE, a propagation context is always returned from a transactional invocation; otherwise a propagation context is only put in a reply message when an OrbixOTS C++ client is invoking on an OrbixOTS C++ server.	FALSE
OTS_NO_OPTIMIZE_PROPAGATION	If set to TRUE, optimization of the propagation context between OrbixOTS C++ clients and servers is turned off; otherwise a smaller context is exchanged using a proprietary GIOP service context. Only relevant for OrbixOTS C++ clients and servers.	FALSE

Performance Tips Summary

The following summarizes the actions you can take to increase the performance of your OrbixOTS applications:

- Use raw disk partitions for transaction logs.
- Take advantage of the group commit feature by increasing the level of concurrency in your servers.
- Reduce the amount of data logged by keeping your application simple: reduce the number of servers participating in transactions; minimize the number of XA resource managers and `CosTransactions::Resource` objects.
- Avoid using transaction log mirrors.
- Avoid using the remote logging feature.
- Reduce unnecessary two-phase commit overheads by using “read-only” transactions.
- Use the XA/one-phase commit optimization.
- If possible use dynamic XA resource manager registration.
- Reduce the overheads associated with the propagation context: reduce the number of servers participating in transactions; create transactions in servers rather than in clients.
- Take advantage of the “allows” transaction policy feature to reduce the number of times transactions need to be propagated to servers.
- Refer to your XA resource manager documentation for performance hints.
- Set the OrbixOTS configuration variable `OTS_NO_OPTIMIZE_PROPAGATION` to `FALSE`.
- Set the OrbixOTS configuration variable `OTS_ALWAYS_RETURN_CONTEXT` to `FALSE`.
- Keep transactions short to avoid tying up shared resources.
- If possible release OCCS locks early (non-strict two-phase locking).

- Tune the user request thread pool to suit your application's needs by setting the OrbixOTS configuration variables `OTS_TPOOL_LWM` and `OTS_TPOOL_HWM`.
- Use `Synchronization` objects to support caching of XA resource manager data.
- If possible, do not wait for heuristics when committing transactions.
- If possible, only use the `Current` interface, and avoid creating unnecessary CORBA objects.
- Experiment with the OrbixOTS configuration variable `OTS_OOB_SYNCHRONOUS`.

Making Software Work Together™

Further Reading

1. IONA Technologies, *Orbix 3.0 Release Notes,* February 1999[1].

2. Object Management Group (OMG), *The Common Object Request Broker: Architecture and Specification, Revision 2.1*, OMG document number 97-09-01, August 1997[2].

3. Object Management Group (OMG), *CORBAservices: Common Object Services Specification,* OMG document number 98-12-09, March 1995.

4. Baker, Seán, *CORBA Distributed Objects: Using Orbix,* Addison-Wesley, November 1997.

5. Henning, Michi, and Steve Vinoski, *Advanced CORBA Programming with C++*, Addison-Wesley, February 1999.

6. Geraghty, Ronan et al., *COM/CORBA Interoperability,* Prentice Hall, January 1999.

7. Slama, Dirk et al., *Enterprise CORBA*, Prentice Hall, March 1999.

[1] Orbix release notes are available from:
`http://www.iona.com/online/support/update/index.html`

[2] OMG documents are available from:
`http://www.omg.org`

Making Software Work Together™

Contact Details

IONA Technologies PLC
The IONA Building
Shelbourne Road
Dublin 4
Ireland
Phone: .... +353 1 637 2000
Fax: .... +353 1 637 2888

IONA Technologies Inc.
200 West St
Waltham, MA 02451
USA
Phone: .... +1 781 902 8000
Fax: .... +1 781 902 8001

IONA Technologies Japan Ltd
Aoyama KK Bldg 7/F
2-26-35 Minami Aoyama
Minato-ku, Tokyo
Japan 107-0062
Phone: .... +813 5771 2161
Fax: .... +813 5771 2162

Support: .... support@iona.com
Training: .... training@iona.com
Orbix Sales: .... sales@iona.com
IONA's FTP site .... ftp.iona.com

World Wide Web: www.iona.com