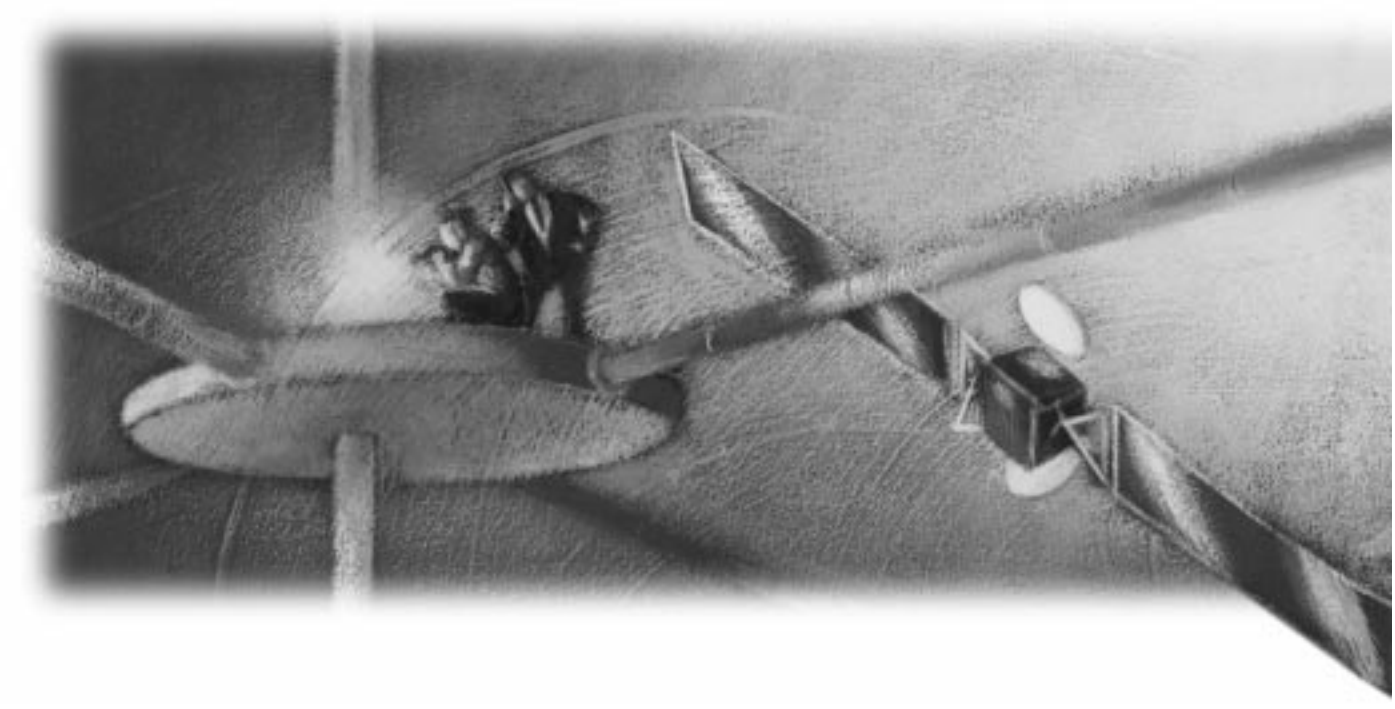


Monitoring Lock Server Performance

Release 6.0

Monitoring Lock Server Performance

Part Number: 60-LSPM-0

Release 6.0, October 30, 2000

Contents

About This Book

This book, *Monitoring Lock-Server Performance*, describes the programming interface for writing C++ programs that gather information about how database applications use a running Objectivity / DB lock server. You can use this information for analyzing performance characteristics of C++, Java or Smalltalk database applications. Monitoring the performance of the lock server may be needed during the following phases of your application: development, tuning, implementation, and post-implementation checking in a production environment

Audience

This book assumes that you are an experienced developer familiar with C++. Database architects may have some use for the conceptual parts of this book.

Organization

Chapter 1 provides some background and considerations when writing a monitoring program.

Chapter 2 is a reference chapter that describes the parts of the programming interface:

- The global names
- Classes providing the functionality
- Classes defining the structure and content of each lock server monitoring message

The appendixes provide examples that illustrate one way of implementing performance monitoring.

Conventions and Abbreviations

Navigation

Table of contents entries, index entries, cross-references, and underlined text are hypertext links.

Typographical Conventions

`oobackup`	Command, literal parameter, code sample, filename, pathname, output on your screen, or Objectivity-defined identifier
`installDir`	Variable element (such as a filename or a parameter) for which you must substitute a value
Browse FD	Graphical user-interface label for a menu item or button
lock server	New term, book title, or emphasized word

Abbreviations

(administration)	Feature intended for database administration tasks
(*FTO*)	Feature of the Objectivity/DB Fault Tolerant Option product
(*DRO*)	Feature of the Objectivity/DB Data Replication Option product
(*IPLS*)	Feature of the Objectivity/DB In-Process Lock Server Option product
(*ODMG*)	Feature conforming to the Object Database Management Group interface

Command Syntax Symbols

[...]	Optional item. You may either enter or omit the enclosed item.
{…}	Item that can be repeated.
...|...	Alternative items. You should enter only one of the items separated by this symbol.
(…)	Logical group of items. The parentheses themselves are not part of the command syntax; do not type them.

Command and Code Conventions

In code examples or commands, the continuation of a long line is indented. Omitted code is indicated with the ellipsis (…) symbol. “Enter” refers to the standard key (labeled either Enter or Return) for terminating a line of input.

Getting Help

We have done our best to make sure all the information you need to install and operate Objectivity products is provided in the product documentation. However, we also realize problems requiring special attention sometimes occur.

Technical Support Web Site

You can find answers to frequently asked questions, supported platforms, known bugs, and bug fixes on the Objectivity Technical Support web site. Send electronic mail or call Objectivity Customer Support to gain access to the site.

How to Reach Objectivity Customer Support

You can contact Objectivity Customer Support by:

- **Telephone:** Call 1.650.254.7100 *or* 1.800.SOS.OBJY (1.800.767.6259) Monday through Friday between 6:00 A.M. and 6:00 P.M. Pacific Time, and ask for Customer Support.

  The toll-free 800 number can be dialed *only* within the 48 contiguous states of the United States and Canada.
- **Fax:** Send a fax to Objectivity at 1.650.254.7171.
- **Electronic Mail:** Send electronic mail to *help@objectivity.com*.

Before You Call

If you need help from Customer Support, please have the following information ready before you contact Objectivity:

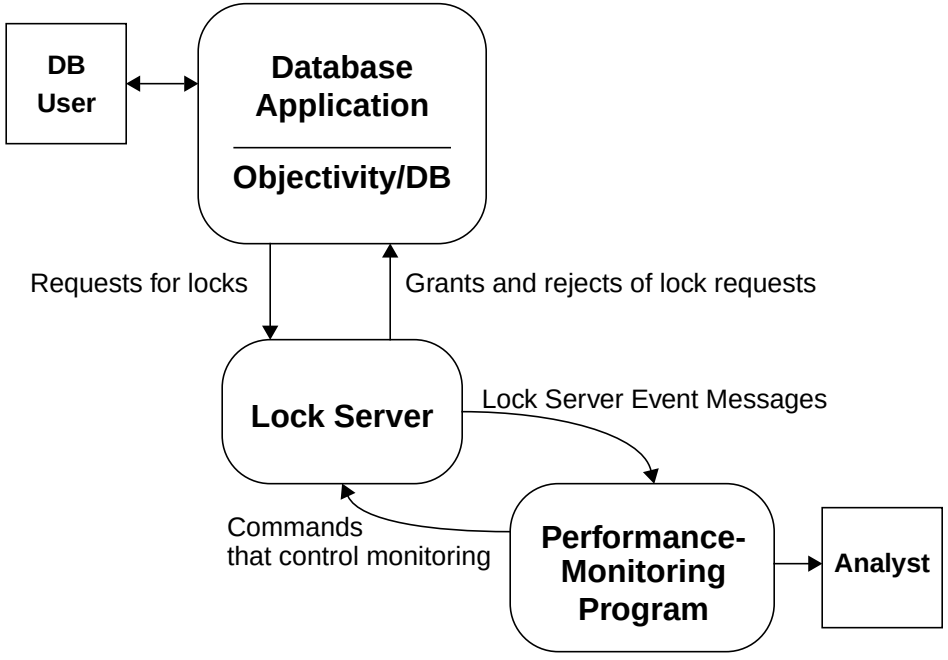
- Your name, company name, address, telephone number, fax number, and email address
- Description of your workstation environment, including the type of workstation, its operating system version, compiler or interpreter, and windowing environment
- Information about the Objectivity product you are using, including the version of the Objectivity / DB libraries
- Detailed description of the problem you have encountered

1

Monitoring Lock Server Performance

The lock server is often the critical component controlling the overall performance of your application. During development, you may need to monitor which clients place a load on the lock server. During production, you may need to know how high the lock server load is and whether the system is about to run into problems. The first step in meeting these needs is monitoring lock server performance. The programming interface documented here enables you to write programs to accomplish that step. In Figure 1-1, such a program is called a *performance-monitoring program*.

Figure 1-1 System View of Monitoring Lock Server Performance



The performance-monitoring program elicits lock server event messages from the lock server. You can analyze these messages and look for ways to make performance improvements in the database application.

Performance-Monitoring Programs

To monitor the performance of the Objectivity / DB lock server, you write a performance-monitoring program that gets messages corresponding to lock server events. These messages are instances of the `oolsEventsMonitorMessage` class, and are called either *event messages* or *message objects* in this book.

A performance-monitoring program sends commands to the lock server to elicit messages from it. The two primary kinds of commands can be categorized as follows:

- *Content commands*—the set of commands that tell the lock server what types of events to monitor:
  - Connects
  - Locks
  - Queues
  - Transactions
  - Deadlocks
- *Activation commands*—the commands that start and stop the monitoring

A performance-monitoring program can monitor any combination of event types at any time. After the event types are chosen and monitoring is activated, the lock server watches for the chosen types of events and, as such events take place, sends the event messages to the process that activated the monitoring.

A performance-monitoring program can send different content commands to change the types of events to be monitored. The program can even request that no events be monitored; in this case, no event messages are sent or received, even if monitoring has been activated.

A performance-monitoring program can ask the lock server to stop monitoring (and therefore stop sending messages). When the lock server stops monitoring, the list of chosen event types is retained—there is no change to or deletion of the list of chosen event types.

In the programming interface, all these commands are provided as member functions of the class `oolsEventsMonitor`:

- Member functions for content commands:
  - `monitorConnects(yes/no)`
  - `monitorLocks(yes/no)`

- monitorQueues(yes/no)
- monitorTransactions(yes/no)
- monitorDeadlocks(yes/no)

- Member functions for activation commands:
  - startMonitoring

    Note: In this book, a program that sends startMonitoring is called the *events monitor*.
  - stopMonitoring

To receive an event message from the lock server, a performance-monitoring program must call the getNextMessage member function on an instance of the oolsEventsMonitor class. This instance of oolsEventsMonitor must be the same object on which startMonitoring was called, because the lock server recognizes that object as the object listening for the messages. (Content commands can be called on any instance of oolsEventsMonitor).

The getNextMessage member function obtains an event message by passing the address of an instance of class oolsEventsMonitorMessage to the lock server; the lock server places the next event message in the message object. A performance-monitoring program must call getNextMessage repeatedly to get all the messages.

The appendixes to this book provide two programs that illustrate how to use the programming interface. Together, the two programs provide the basic functionality needed for lock server monitoring.

Designing

This section discusses factors to take into consideration when designing programs for lock-server performance monitoring.

Supported Language

This programming interface supports C++ programs only.

Header File

The classes `oolsEventsMonitorMessage`, `oolsEventsMonitor`, and other required classes are in the header file `oolspm.h` which must be included by the programs. For more information, see the installed file.

UDP

This programming interface uses User Datagram Protocol (UDP) sockets to send and receive messages. Because of the characteristics—no congestion avoidance, no packet delivery guarantees, fragmentation problems—of UDP sockets, some messages may be lost, received out of sequence, or received more than once. As the designer of the performance-monitoring programs, you should take these possibilities into consideration.

Implicit Monitoring

Whenever transactions are monitored, there is an implicit monitoring of connects. Moreover, whenever locks are monitored, there is an implicit monitoring of transactions. Therefore, whenever the program invokes `monitorLocks`, it implicitly invokes `monitorTransactions` and `monitorConnects`.

Any explicit monitoring of implicitly monitored events would still continue after switching off monitoring of lock or transaction events.

Number of Programs

The number of programs and the types of user interface are dictated by the needs of the analysis you want to perform. In the simplest case, you can create a single program that performs all monitoring tasks. For more flexibility, you could use a separate driver program to request the types of events to be monitored (see Appendix A); to complement the driver program, a separate monitor program would start the monitoring and receive the messages, and, for example, display them dynamically and store them as needed in a file for later analysis. In that case, you may want a third program to analyze the stored messages.

NOTE Only the object that starts monitoring (the events monitor) can receive messages.

Required Tasks

The program or programs must handle the following tasks:

- You may request monitoring of one or many types of events—or none—as required and at any time.
- The events monitor, an `oolsEventsMonitor` object, calls `startMonitoring`.
- When the lock server generates messages of the types that you have requested, the events monitor receives the messages.
- The messages are processed as required until you no longer need monitoring.
- If the lock server is killed or if a process calls `stopMonitoring`, then the lock server sends `oolsLspmStoppedMessage`. The events monitor should handle this special message and not wait for any more new messages.

The events monitor must instantiate the following classes:

- `oolsEventsMonitor`
- `oolsEventsMonitorMessage`

Message Sequence

When the lock server receives a `startMonitoring` command:

- The lock server starts sending messages through a UDP socket to the event monitor as the requested events occur.
- The events monitor's `getNextMessage` member function passes the address of the message object.
- Each call to `getNextMessage` picks up one message as the message arrives at the UDP socket, whereupon `getNextMessage` returns the message to the event monitor by filling the message object with messages received from the lock server.

Appendix B, "Monitor Program Sample," illustrates a way to implement this.

Obtaining a Corroborating Transaction Identifier

Obtaining specific transactions' identifiers from within the database application can help corroborate monitoring data. The transaction class has a member function `getID` that returns the transaction identifier assigned by the lock server to the current transaction:

```
ooTransId ooTrans::getID()
```

NOTE The `getID` member function is used from within the database application itself and not from within any of the monitoring programs you write.

Relation of Message-Object Values and Mnemonics

Table 1-1 lists the number and mnemonic for the messages that the events monitor can receive as enabled by the event selecting member function.

Table 1-1: Message Values and Mnemonics for Message Objects

Event Selecting Member Function	Message Object		Page
	Message Type Value	Mnemonic	
monitorLocks	1	lockMsg	46
	2	unlockMsg	49
	3	btMsg[a]	39
	4	etMsg	45
	5	downgradeMsg	44
	11	connectMsg[a]	41
	12	disconnectMsg	43
monitorTransactions	3	btMsg[a]	39
	4	etMsg	45
	5	downgradeMsg	44
	11	connectMsg[a]	41
	12	disconnectMsg	43
monitorConnects	11	connectMsg[a]	41
	12	disconnectMsg	43
monitorDeadlocks	8	deadlockFirstMsg	42
	9	deadlockMsg	43
monitorQueues	6	waitFailedMsg	50
	7	waitSucceededMsg	51

a. Hostnames can be reported differently in the messages involving `oolsconnectMessage` and `oolsbeginTransactionMessage`.

Linking a Program

A performance-monitoring program needs the libraries as shown for the respective platform.

Windows

Table 1-2 lists the import libraries you can choose for linking a lock-server performance-monitoring program. The release or debug import library is linked automatically when you include the `oolspm.h` header file.

Table 1-2: Dynamic Link Import Libraries

Library File	Description
`oolspmi.lib`	Multithreaded import library
`oolspmid.lib`	Debug version of the multithreaded import library

The program should also link to the appropriate Objectivity / DB library: `oodbi.lib` or `oodbid.lib`.

Table 1-3 lists the dynamic link libraries (DLLs) that must be available at runtime.

Table 1-3: Dynamic Link Libraries

DLL File	Description
`oolspm`*xx*`.dll`[a]	Dynamic link library
`oolspm`*xx*`d.dll`[a]	Debug version of the dynamic link library

a. The digits *xx* in this DLL name correspond to the current Objectivity/DB release.

For more information, see Appendix A, "C++ Application Development," of *Installation and Platform Notes for Windows*.

UNIX

On UNIX platforms, you can link a performance-monitoring program with static or shared libraries.

For static linking, link your program to both of the following static libraries:

- `liboolspm.a`
- `liboo.a`

For dynamic linking, shared libraries have naming conventions based on the platform. For more information, see Appendix A, “C++ Application Development,” of *Installation and Platform Notes for UNIX*.

2

Programming Interface

The programming interface for programs monitoring lock server performance is designed for low-overhead programs in C++. These programs are not database applications and so do not use `oo.h`.

Class names in the interface begin with `ools`.

Global Names

This section lists global types in alphabetical order.

oolsLockResult

global type

The result of the lock event can get any of these values.

Constants

`lkeror`
: There was an error with the lock event.

`granted`
: The lock was granted.

`upgraded`
: The lock was upgraded.

`queued`
: The lock request was queued.

`deadlock`
: A deadlock condition was detected.

`rejected`
: The lock request was rejected.

`probeSucceeded`
: This result reflects an internal state.

`probeFailed`
: This result reflects an internal state.

oolsLockType

global type

The type of lock.

Constants

`noLock`

0 = There is no lock.

`IS`

1 = This type reflects an internal state.

`IC`

2 = This type reflects an internal state.

`IX`

6 = This type reflects an internal state.

`S`

3 = This is an MROW read.

`R`

4 = This is a read lock.

`C`

5 = This is a change lock.

`X`

7 = This is an exclusive lock.

oolsMessageType

global type

The type that denotes the content of the message. This is used for the `msgType` field of a message object. The `msgType` field determines which variant of the union field `msg` has the message.

Constants

`noMessageType`

`lockMessageType`

`unlockMessageType`

`beginTransactionMessageType`

`endTransactionMessageType`

`downgradeMessageType`

`waitFailedMessageType`

`waitSucceededMessageType`

`deadlockFirstMessageType`

`deadlockMessageType`

`deadlockLastMessageType`

`connectMessageType`

`disconnectMessageType`

`lspmStoppedMessageType`

Discussion

Hostnames can be reported differently in the messages involving `connectMessageType` and `beginTransactionMessageType`.

oolsResourceType

global type

The level of resource involved in the event.

Constants

`FDB`

The federated database level.

`DB`

The database level.

`OC`

The container level.

Discussion

`oolsResourceType` tells which level of entity the message refers to. In the hierarchy of levels for these entities, your program must ignore the identifier numbers below the level of the resource.

oolsResult

global type

The result for the event.

Constants

`error`

The requesting application received an error.

`succeeded`

The application's requests succeeded.

oolsStopCause

global type

The source of the stop of the lock-server monitoring.

Constants

`lsKilled`

The lock server process was killed.

`userStop`

A user stopped the lock sever monitoring through the member function `stopMonitoring`.

oolsTimestamp

global type

The date and time of the event.

oolsTransEndType

global type

The type of end transaction.

Constants

`abortTran`

An abort was issued for the transaction.

`commitTran`

A commit was issued for the transaction.

`releaseLatchesTran`

This represents an internal type.

`downgradeTran`

A commitAndHold with the downgrade option was issued for the transaction.

oolsTransId

global type

The transaction identifier. For additional information on transaction identifiers, refer to “Obtaining a Corroborating Transaction Identifier” on page 13.

YN

global type

For member functions checking monitoring and returning `yn`; for member functions selecting monitoring by specifying `yn`. Implemented as an unsigned integer.

Constants

`yes = 0xFFFFFFFF`

`no = 0`

oolsEventsMonitor Class

The class `oolsEventsMonitor` represents an *events monitor*. More precisely, the events monitor is the one and only object that calls the `startMonitoring` and `getNextMessage` member functions of `oolsEventsMonitor`.

The remaining member functions for this class fall into four categories:

- Telling the Lock Server the Events to Monitor

  These setter member functions select the events that will be monitored.
- Testing Event Selection

  These getter member functions correspond to the setter member functions in the Telling the Lock Server the Events to Monitor category. These methods identify the list of events currently under monitoring in the lock server. The information can be obtained if monitoring is started or stopped.
- Checking Lock Server Monitoring
- Stopping Lock Server Monitoring

Telling the lock server what to monitor occurs usually before monitoring is started. However, the events to monitor can be changed dynamically at any time. When none of the events is being monitored, even though monitoring may still be in effect, no message is sent with the possible exception of `oolsLspmStoppedMessage`.

Reference Summary

Category	Issued From	Member Function
Telling the Lock Server the Events to Monitor	Any process Any object	monitorConnects monitorDeadlocks monitorLocks monitorQueues monitorTransactions
Monitoring Events	The events monitor object	getNextMessage startMonitoring
Testing Event Selection	Any process Any object	isMonitoringConnects isMonitoringDeadlocks isMonitoringLocks isMonitoringQueues isMonitoringTransactions
Checking Lock Server Monitoring	Any process Any object	isMonitoringActive
Stopping Lock Server Monitoring	Any process Any object	stopMonitoring

Reference Index

getNextMessage	Delivers the next message received from the lock server.
isMonitoringActive	Asks the lock server if there is a process which is monitoring at this time, irrespective of any types of events being monitored
isMonitoringConnects	Asks the lock server if connect events will be monitored.
isMonitoringDeadlocks	Asks the lock server if deadlock events will be monitored.
isMonitoringLocks	Asks the lock server if lock events will be monitored.
isMonitoringQueues	Asks the lock server if queue events will be monitored.

isMonitoringTransactions	Asks the lock server if transaction events will be monitored.
monitorConnects	Adds or removes connect events from the list of monitored events.
monitorDeadlocks	Adds or removes deadlock events from the list of monitored events.
monitorLocks	Adds or removes lock events from the list of monitored events.
monitorQueues	Adds or removes queue events from the list of monitored events.
monitorTransactions	Adds or removes transaction events from the list of monitored events.
oolsEventsMonitor	Constructs an events monitor.
startMonitoring	Establishes a connection with the lock server and identifies the current process as the events monitor, the process to which the lock server is to send event-monitoring messages.
stopMonitoring	Causes the lock server to stop, disconnect, and send a last message.

Constructors

oolsEventsMonitor

Constructs an events monitor.

```
oolsEventsMonitor(const char *h);
```

Parameters

h

Lock server hostname whose events will be monitored.

Discussion

The constructor holds the specified server hostname to which a connection is established. The hostname is used when any method is used.

`oolsEventsMonitor` is used to control monitoring. After creation of an instance, the program can, as needed, invoke member functions of the class to affect the monitoring dynamically. The `getNextMessage` member function takes an instance of the class `oolsEventsMonitorMessage`, and the instance is filled with the data received from the lock server.

The constructor throws the following exceptions:

- `olspmLockServerNotAvailableException`
- `oolspmLockServerConnectionClosedException`

Member Functions

getNextMessage

Delivers the next message received from the lock server.

```
int getNextMessage(
     oolsEventsMonitorMessage  *msg,
     int msecWaitTime = 0);
```

Parameters

msg

The address of the message in which to return the next message from the lock server.

msecWaitTime

If the next message is not received in *msecWaitTime* milliseconds, the call will return with an error.

Returns

1 if there is a message; 0 if no message was collected within the time limit set by *msecWaitTime*.

Discussion

This member function reads from the socket that was created by the call to `startMonitoring`.

Use `msgNumber` in *`*msg`* to determine that messages are

- Missing
- Duplicates
- Out of sequence

Use `msgType` to determine the appropriateness of processing a union member.

Do not issue any more `getNextMessage` calls or wait for any more new messages after receiving a `lspmStoppedMessage` type of message, because the lock server has stopped sending messages and has disconnected itself from the socket given to the process running your program. If the lock server is killed, `cause` will be `lskilled`.

isMonitoringActive

Asks the lock server if there is a process which is monitoring at this time, irrespective of any types of events being monitored

```
YN isMonitoringActive(void);
```

Returns `yes` if monitoring is active; otherwise, `no`, if monitoring is not active.

Discussion `isMonitoringActive` throws the following exceptions:

- `oolspmLockServerNotAvailableException`
- `oolspmLockServerConnectionClosedException`
- `oolspmUnknownException`

isMonitoringConnects

Asks the lock server if connect events will be monitored.

```
YN isMonitoringConnects(void);
```

Returns `yes`, if connects are monitored; otherwise, `no`, if connects are not monitored.

Discussion `isMonitoringConnects` throws the following exceptions:

- `oolspmLockServerNotAvailableException`
- `oolspmLockServerConnectionClosedException`
- `oolspmUnknownException`

isMonitoringDeadlocks

Asks the lock server if deadlock events will be monitored.

```
YN isMonitoringDeadlocks(void);
```

Returns `yes`, if deadlocks are monitored; otherwise, `no`, if deadlocks are not monitored.

Discussion `isMonitoringDeadlocks` throws the following exceptions:

- `oolspmLockServerNotAvailableException`
- `oolspmLockServerConnectionClosedException`
- `oolspmUnknownException`

isMonitoringLocks

Asks the lock server if lock events will be monitored.

```
YN isMonitoringLocks(void);
```

Returns yes, if lock, transaction, and connect events are monitored; otherwise, no, if lock events are not monitored. If no is returned, further checking is required for the status of monitoring transaction and connect events.

Discussion `isMonitoringLocks` throws the following exceptions:

- `oolspmLockServerNotAvailableException`
- `oolspmLockServerConnectionClosedException`
- `oolspmUnknownException`

isMonitoringQueues

Asks the lock server if queue events will be monitored.

```
YN isMonitoringQueues(void);
```

Returns yes, if queue events are monitored; otherwise, no, if queue events are not monitored.

Discussion `isMonitoringQueues` throws the following exceptions:

- `oolspmLockServerNotAvailableException`
- `oolspmLockServerConnectionClosedException`
- `oolspmUnknownException`

isMonitoringTransactions

Asks the lock server if transaction events will be monitored.

```
YN isMonitoringTransactions(void);
```

Returns yes, if transaction and connect events are monitored; otherwise, no, if transaction events are not monitored. If no is returned, further checking is required for the status of monitoring connect events.

Discussion `isMonitoringTransactions` throws the following exceptions:

- `oolspmLockServerNotAvailableException`
- `oolspmLockServerConnectionClosedException`
- `oolspmUnknownException`

monitorConnects

Adds or removes connect events from the list of monitored events.

```
void monitorConnects (YN yn = yes);
```

Parameters *yn*

yes	Add connect events.
no	Remove connect events.

Discussion If *yn* is yes, `monitorConnects` enables sending of the following messages:

- `oolsConnectMessage`
- `oolsDisconnectMessage`

`monitorConnects` throws the following exceptions:

- `oolspmLockServerNotAvailableException`
- `oolspmLockServerConnectionClosedException`
- `oolspmUnknownException`

monitorDeadlocks

Adds or removes deadlock events from the list of monitored events.

```
void monitorDeadlocks (YN yn = yes);
```

Parameters *yn*

yes	Add deadlock events.
no	Remove deadlock events.

Discussion If *yn* is yes, `monitorDeadlocks` enables sending of the following messages:

- `oolsDeadLockFirstMessage`
- `oolsDeadLockMessage`

`monitorDeadlocks` throws the following exceptions:

- `oolspmLockServerNotAvailableException`
- `oolspmLockServerConnectionClosedException`
- `oolspmUnknownException`

monitorLocks

Adds or removes lock events from the list of monitored events.

```
void monitorLocks (YN yn = yes);
```

Parameters *yn*

yes	Add lock events.
no	Remove lock events.

Discussion If *yn* is yes, `monitorLocks` enables sending of the following types of messages:

- `oolsLockMessage`
- `oolsUnlockMessage`
- `oolsDowngradeMessage`

`monitorLocks` implicitly monitors both transactions and connects. Take this into account when dealing with downgrade lock messages of the `oolsDowngradeMessage` type.

`monitorLocks` throws the following exceptions:

- `oolspmLockServerNotAvailableException`
- `oolspmLockServerConnectionClosedException`
- `oolspmUnknownException`

monitorQueues

Adds or removes queue events from the list of monitored events.

```
void monitorQueues (YN yn = yes);
```

Parameters *yn*

yes	Add queue events.
no	Remove queue events.

Discussion If *yn* is yes, `monitorQueues` enables sending of the following messages:

- `oolsWaitFailedMessage`
- `oolsWaitSuccededMessage`

`monitorQueues` throws the following exceptions:

- `oolspmLockServerNotAvailableException`
- `oolspmLockServerConnectionClosedException`
- `oolspmUnknownException`

monitorTransactions

Adds or removes transaction events from the list of monitored events.

```
void monitorTransactions (YN yn = yes);
```

Parameters *yn*

yes	Add transaction events.
no	Remove transaction events.

Discussion If *yn* is yes, `monitorTransactions` enables sending of the following messages:

- `oolsBeginTransaction`
- `oolsEndTransaction`
- `oolsDowngradeMessage`

`monitorTransactions` implicitly monitors connects.

`monitorTransactions` throws the following exceptions:

- `oolspmLockServerNotAvailableException`
- `oolspmLockServerConnectionClosedException`
- `oolspmUnknownException`

startMonitoring

Establishes a connection with the lock server and identifies the current process as the events monitor, the process to which the lock server is to send event-monitoring messages.

```
void startMonitoring();
```

Discussion If the program requests monitoring of any of the five types of events, then, as events occur, corresponding messages are sent to this monitoring process.

Use `monitorLocks` and the other setter member functions at any time and from any process or object to change the list of event types monitored. Thus, the monitoring can be dynamically changed.

`startMonitoring` throws the following exceptions:

- `oolspmLockServerNotAvailableException`
- `oolspmLockServerConnectionClosedException`
- `oolspmCantConnectToLockServerException`
- `oolspmLockServerMonitoringAlreadyStartedException`

stopMonitoring

Causes the lock server to stop, disconnect, and send a last message.

```
void stopMonitoring ();
```

Discussion

The sequence of events is the following:

- Stop monitoring any type of event
- Disconnect from the initiating process
- Send one last message of type `lpsmStoppedMessageType`

The list of types to be monitored is not forgotten by the lock server.

`stopMonitoring` throws the following exceptions:

- `oolspmLockServerNotAvailableException`
- `oolspmLockServerConnectionClosedException`
- `oolspmLockServerMonitorNotRunningException`

The special message `lpsmStoppedMessageType` is sent by the lock server when it is killed or when a user explicitly stops monitoring. The field `cause` identifies which kind of event triggered the message.

oolsEventsMonitorMessage Class

A message object is an instance of `oolsEventsMonitorMessage`. A message object contains one of the set of possible messages that the events monitor places in the variable `msg`.

Data Members

msg

The particular message in this message object.

```
union
{
     oolsLockMessage lockMsg;
     oolsUnlockMessage unlockMsg;
     oolsBeginTransactionMessage btMsg;
     oolsEndTransactionMessage etMsg;
     oolsDowngradeMessage downgradeMsg;
     oolsWaitFailedMessage waitFailedMsg;
     oolsWaitSucceededMessage waitSucceededMsg;
     oolsDeadlockFirstMessage deadlockFirstMsg;
     oolsDeadlockMessage deadlockMsg;
     oolsDeadlockLastMessage deadlockLastMsg;
     oolsConnectMessage connectMsg;
     oolsDisconnectMessage disconnectMsg;
     oolsLspmStoppedMessage lspmStoppedMsg
} msg;
```

Discussion

A `msg` of a particular type will be received if any one of the event types is being monitored that causes, explicitly or implicitly, such messages to be sent. See Table 1-1, “Message Values and Mnemonics for Message Objects,” on page 14.

You can use the `msgType` data member to find out the type of message in `msg`.

msgNumber

A message identifier issued by the lock server.

```
unsigned int msgNumber;
```

msgType

The type of lock server event described by the message in the `msg` data member.

```
oolsMessageType msgType;
```

Discussion

When `msgType` is 0, there is no message to process. Values between 1 and 13 stand for the classes in `msg` in the order shown. For example, when `msgType` is 1, `lockMsg` contains the message, and when `msgType` is 13, `lspmStoppedMsg` contains the message. For more information, refer to Table 1-1, “Message Values and Mnemonics for Message Objects,” on page 14.

Member Functions

save

Saves the message to the specified file.

```
void save(FILE *msgFile);
```

Discussion

The `save` member function is a placeholder only, and no implementation is provided in the library. For an illustrative example of an implementation, see the file `monitor.C` and Appendix B, “Monitor Program Sample”.

Message Classes

This chapter lists, in alphabetical order, the classes that describe the data for each type of message which the lock server can provide. These classes appear in a union of the message classes within the message object.

The message classes can by categorized by the member functions that request listening for the events the messages describe. These member functions and the messages are listed in the following table.

Within some of these message classes are the following classes:

- `oolsClientId`
- `oolsResourceId`

Within `oolsResourceId` and the message classes are the following `typedefs`, explained in “Global Names” on page 19:

- `oolsMessageType`
- `oolsStopCause`
- `oolsLockResult`
- `oolsResult`
- `oolsLockType`
- `oolsTransEndType`
- `oolsResourceType`

Data members are explained, sometimes redundantly, in each class description. Each class has a `save` member function which your program can implement as required. It is mainly intended to help you write to a file. For completeness, each class description also shows the `save` member function as it is declared in the class.

Reference Index

oolsWaitFailedMessage	Describes a lock request that was queued and waiting and was subsequently refused and deleted from the queue, giving a timestamp, the type of lock, and ids for the transaction, resource, and user.
oolsWaitSucceededMessage	Describes a successful granting of a waiting lock request, giving a timestamp, the time waiting in the queue, the type of lock, and identifiers for the transaction, resource, and client.

Class Definitions

oolsBeginTransactionMessage

Describes a the lock server's handling of a begin transaction event.

```
// Data Members
oolsTimestamp ts;
oolsTransId tid; /* will be 0 in case of error */
unsigned int apid;
unsigned int fdid;
char fdName[MAXFDNAME];
oolsClientId client;
oolsResult result;

// Member Functions
void save(FILE *msgFile) ;
```

Data Member

`ts`

Timestamps as provided by the operating system showing the time when the event took place.

`tid`

Identifier of the transaction on whose behalf events are taking place. A transaction identifier is assigned only at the time of opening a federated database and not at the time transaction start is issued. The member function `ooTransId ooTrans::getID()` returns the transaction identifier assigned by the lock server to the current transaction.

In case of an error, `tid` is 0.

`apid`

The autonomous partition's identifier assigned by the `oonewap` tool. 65535 is the identifier of the federated database.

`fdid`

The federated-database number specified by the `oonewfd` tool.

`fdname`

The federated database name, limited to 1024 characters.

`client`

The client identifier. See the class `oolsClientId`. If the client runs on Windows NT, the hostname reported in `oolsConnectMessage` can be different from the hostname reported in this `oolsBeginTransactionMessage`; this discrepancy can occur if the functions `getpeername(socket)` and `hostname()` return different values.

`result`

The result returned to the application that issued the begin transaction. See the global `oolsLockResult`.

Member Function

`save`

The `save` member function is a placeholder only, and no implementation is provided in the library. For an illustrative example of an implementation, see the file `monitor.C` and Appendix B, "Monitor Program Sample".

oolsClientId

Identifies the client by user, process, and host identifiers.

```
// Data Members
int uid;
int pid;
char host[MAXHOSTNAMELENGTH];

// Member Functions
void save(FILE *msgFile) ;
```

Data Member `uid`

The user identifier of the process on whose behalf the lock server event is taking place. For clients using UNIX, this is what `getuid()` gives; for clients

using Windows NT, this is a number assigned by Objectivity. In either case, it is the user identifier that `oolockmon` shows.

`pid`

The process identifier. Unused for lock messages.

`host`

The name of the host machine where the client is running.

The largest number of characters in the name, `MAXHOSTNAMELENGTH`, is 256.

Member Function

`save`

The `save` member function is a placeholder only, and no implementation is provided in the library. For an illustrative example of an implementation, see the file `monitor.C` and Appendix B, “Monitor Program Sample”.

oolsConnectMessage

Describes a connection with a timestamp, client identifier, the socket, and the result of the connection.

```
// Data Members
oolsTimestamp ts;
oolsClientId client;
oolsSocket sock;
oolsResult result;

// Member Functions
void save(FILE *msgFile);
```

Data Member

`ts`

Timestamps as provided by the operating system, showing the time when the event took place.

`client`

The client identifier. See the class `oolsClientId`. If the client runs on Windows NT, the hostname reported in `oolsBeginTransactionMessage` can be different from the hostname reported in this `oolsConnectMessage`; this discrepancy can occur if the functions `getpeername(socket)` and `hostname()` return different values.

`sock`

The socket the lock server assigned for receiving requests from a client. If `OO_NT` is defined, then sock is a `SOCKET` as defined in `winsock2.h`; otherwise, sock is an `int`.

`result`

The result returned to the application that issued the connect. See the global `oolsLockResult`.

Member Function

`save`

The `save` member function is a placeholder only, and no implementation is provided in the library. For an illustrative example of an implementation, see the file `monitor.C` and Appendix B, “Monitor Program Sample”.

oolsDeadlockFirstMessage

Signals the start of a deadlock chain, describing the first transaction in a deadlock situation with a timestamp and the transaction identifier.

```
// Data Members
oolsTimestamp ts;
oolsTransId tid;

// Member Functions
void save(FILE *msgFile) ;
```

Data Member

`ts`

Timestamps as provided by the operating system, showing the time when the event took place.

`tid`

Identifier of the transaction on whose behalf events are taking place.

Member Function

`save`

The `save` member function is a placeholder only, and no implementation is provided in the library. For an illustrative example of an implementation, see the file `monitor.C` and Appendix B, “Monitor Program Sample”.

oolsDeadlockMessage

Provides one of a set of transaction identifiers involved in a deadlock situation.

```
// Data Members
oolsTransId tid;

//Member Functions
void save(FILE *msgFile) ;
```

Data Member `tid`

Identifier of a transaction that is waiting for a lock.

Member Function

`save`

The `save` member function is a placeholder only, and no implementation is provided in the library. For an illustrative example of an implementation, see the file `monitor.C` and Appendix B, “Monitor Program Sample”.

Discussion Each of the set of deadlock messages carries a separate `msgNumber`. The `tid` is a common attribute between the elements of a set of messages for a deadlock.

oolsDisconnectMessage

Describes a disconnect with a timestamp and the socket assigned by the lock server at connect time.

```
// Data Members
oolsTimestamp ts;
oolsSocket sock;

// Member Functions
void save(FILE *msgFile);
```

Data Member `ts`

Timestamps as provided by the operating system, showing the time when the event took place.

`sock`

The socket the lock server assigned for receiving requests from a client when the connection was made. If `OO_NT` is defined, then `sock` is a `SOCKET` as defined in `winsock2.h`; otherwise, `sock` is an `int`.

Member Function

`save`

The `save` member function is a placeholder only, and no implementation is provided in the library. For an illustrative example of an implementation, see the file `monitor.C` and Appendix B, “Monitor Program Sample”.

Discussion The connect time is described in a connect message, `oolsConnectMessage`.

---

oolsDowngradeMessage

Describes the lock server’s downgrading of the lock requests within a transaction from update to read.

```
// Data Members
oolsTimestamp ts;
oolsTransId tid;
oolsResult result;

// Member Functions
void save(FILE *msgFile) ;
```

Data Member `ts`

Timestamps as provided by the operating system, showing the time when the event took place.

`tid`

Identifier of the transaction on whose behalf events are taking place.

`result`

The result returned to the application that issued the downgrade. See the global `oolsLockResult`.

Member Function

`save`

The `save` member function is a placeholder only, and no implementation is provided in the library. For an illustrative example of an implementation, see the file `monitor.C` and Appendix B, “Monitor Program Sample”.

Discussion A downgrade is the result of the application submitting a commitAndHold request with the downgrade option. This affects all the locks held by the transaction.

oolsEndTransactionMessage

Describes the lock server's handling of a commit, abort or commitAndHold request.

```
// Data Members
oolsTimestamp ts;
oolsTransId tid;
unsigned int apid;
oolsTransEndType type;
oolsResult result;

// Member Functions
void save(FILE *msgFile) ;
```

Data Member

`ts`

Timestamps as provided by the operating system, showing the time when the event took place.

`tid`

Identifier of the transaction on whose behalf events are taking place.

In case of an error, `tid` is 0.

`apid`

The autonomous partition's identifier assigned by the `oonewap` tool. 65535 is the identifier of the federated database.

`type`

The type of end transaction. See the global `oolsTransEndType`.

`result`

The result returned to the application that issued the commit, abort or commitAndHold. See the global `oolsLockResult`.

Member Function

`save`

The `save` member function is a placeholder only, and no implementation is provided in the library. For an illustrative example of an implementation, see the file `monitor.C` and Appendix B, “Monitor Program Sample”.

oolsLockMessage

Describes a lock event with a result and a detail consisting of a type, a timestamp, and identifiers for transaction, resource, and client.

```
// Data Members
oolsTimestamp ts;
oolsTransId tid;
oolsResourceId resrc;
oolsLockType mode;
oolsClientId client;
oolsLockResult result;

// Member Functions
void save(FILE *msgFile) ;
```

Data Member

`ts`

Timestamps as provided by the operating system, showing the time when the event took place.

`tid`

Identifier of the transaction on whose behalf events are taking place.

`resrc`

The resource identifier. See the class `oolsResourceId`.

`mode`

The type of lock. See the global `oolsLockType`.

`client`

The client identifier. See the class `oolsClientId`. In this context, `host` and `pid` are not used and have the respective values `0` and null string.

`result`

The result returned to the application that requested the lock. See the global `oolsLockResult`.

Member Function

`save`

The `save` member function is a placeholder only, and no implementation is provided in the library. For an illustrative example of an implementation, see the file `monitor.C` and Appendix B, “Monitor Program Sample”.

---

oolsLspmStoppedMessage

Provides the time and source for the stopping of lock server monitoring.

```
// Data Members
oolsTimestamp ts;
oolsStopCause cause;

// Member Functions
void save(FILE *msgFile);
```

Data Member `ts`

Timestamps as provided by the operating system, showing the time when the event took place.

`cause`

The source for the stopping of lock server monitoring. See the global `oolsStopCause`.

Member Function

`save`

The `save` member function is a placeholder only, and no implementation is provided in the library. For an illustrative example of an implementation, see the file `monitor.C` and Appendix B, “Monitor Program Sample”.

oolsResourceId

Describes a resource by its level in the storage hierarchy and by the identifiers of its partition, federated database, database, container, and version (currently fixed at 0).

```
// Data Members
oolsResourceType resource;
unsigned int apid;
unsigned int fdid;
unsigned int dbid;
unsigned int ocid;
unsigned int versn;

// Member Functions
void save(FILE *msgFile) ;
```

Data Member

`resource`

The resource type. See the global `oolsResourceType`.

`apid`

The autonomous partition's identifier assigned by the `oonewap` tool. 65535 is the identifier of the federated database.

`fdid`

The federated-database number specified by the `oonewfd` tool.

`dbid`

The identifier assigned to the database during `oonewdb`. Database identifiers of 0 and 1 are internal to Objectivity.

`ocid`

The identifier assigned to a container when it is created.

Internal to Objectivity are a number of `ocids`: 0, 1, 32767 and above, and a few others. Subtract 32767 from an `ocid` greater than 32767 to get the identifier of a user-created container.

`versn`

Currently, this is always 0. It is intended for the version number of the container identified in the message.

Member Function

`save`

The `save` member function is a placeholder only, and no implementation is provided in the library. For an illustrative example of an implementation, see the file `monitor.C` and Appendix B, “Monitor Program Sample”.

oolsUnlockMessage

Describes an unlock event with a timestamp, the result and identifiers for the transaction and resource.

```
// Data Members
oolsTimestamp ts;
oolsTransId tid;
oolsResourceId resrc;
oolsResult result;

// Member Functions
void save(FILE *msgFile) ;
```

Data Member

`ts`

Timestamps as provided by the operating system, showing the time when the event took place.

`tid`

Identifier of the transaction on whose behalf events are taking place.

`resrc`

The resource identifier. See the class `oolsResourceId`.

`result`

The result returned to the application that requested the lock. See the global `oolsLockResult`.

Member Function

`save`

The `save` member function is a placeholder only, and no implementation is provided in the library. For an illustrative example of an implementation, see the file `monitor.C` and Appendix B, “Monitor Program Sample”.

Discussion When a transaction ends, all the locks held at the time are released. There is no unlock message sent to reflect the release. The unlock is implicit, and there is no unlock message sent for any individual lock.

oolsWaitFailedMessage

Describes a lock request that was queued and waiting and was subsequently refused and deleted from the queue, giving a timestamp, the type of lock, and ids for the transaction, resource, and user.

```
// Data Members
oolsTimestamp ts;
oolsTransId tid;
oolsResourceId resrc;
oolsLockType mode;
int uid;

// Member Functions
void save(FILE *msgFile) ;
```

Data Member `ts`

Timestamps as provided by the operating system, showing the time when the event took place.

`tid`

Identifier of the transaction on whose behalf events are taking place.

`resrc`

The resource identifier. See the class `oolsResourceId`.

`mode`

The type of lock that the transaction is waiting for. See the global `oolsLockType`.

`uid`

The user identifier of the process on whose behalf the lock server event is taking place. For clients using UNIX, this is what `getuid()` gives; for clients using Windows NT, this is a number assigned by Objectivity. In either case, it is the user identifier that `oolockmon` shows. This is the same `uid` in the begin transaction message `oolsBeginTransactionMessage`.

Member Function

`save`

The `save` member function is a placeholder only, and no implementation is provided in the library. For an illustrative example of an implementation, see the file `monitor.C` and Appendix B, “Monitor Program Sample”.

oolsWaitSucceededMessage

Describes a successful granting of a waiting lock request, giving a timestamp, the time waiting in the queue, the type of lock, and identifiers for the transaction, resource, and client.

```
// Data Members
oolsTimestamp ts;
oolsTransId tid;
oolsResourceId resrc;
oolsLockType mode;
oolsClientId client;

unsigned int timeInQ;

// Member Functions
void save(FILE *msgFile);
```

Data Member

`ts`

Timestamps as provided by the operating system, showing the time when the event took place.

`tid`

Identifier of the transaction on whose behalf events are taking place.

`resrc`

The resource identifier. See the class `oolsResourceId`.

`mode`

The type of lock. See the global `oolsLockType`.

`client`

The client identifier. See the class `oolsClientId`. In this context, `host` and `pid` are not used and have the respective values 0 and null string.

`timeInQ`

The time, in seconds, that the lock request waited in the queue before the lock server granted it.

Member Function

`save`

The `save` member function is a placeholder only, and no implementation is provided in the library. For an illustrative example of an implementation, see the file `monitor.C` and Appendix B, “Monitor Program Sample”.

A

Driver Program Sample

The following listing provides a sample program that enables you to perform the following tasks:

- Control the list of events to monitor
- Query the lock server for its state

NOTE Define the appropriate platform name before trying to compile the program.

```
//#define OO_NT
#define OO_SOLARIS

// lspm  driver -
// This program is the driver to send requests to lock server
// in order to control what to monitor.
// This program cannot receive monitoring messages.

#include "oolspm.h"
#include <stdlib.h>


int main(int argc, char* argv[])

{
     oolsEventsMonitor *mon;

     char host[MAXHOSTNAMELENGTH];
     char eventStr[80];
     int event;
     YN stat;


     printf("What is the hostname of the lock server to monitor?\n");
     scanf("%s", host);
```

```
     printf("[driver.C]Going to use lock server at: %s.\n", host);

     try
     {
     mon = new oolsEventsMonitor(host);
     }
     catch (oolspmException &ex)
     {
     printf ("%s.\n", ex.what());
     return 1;
     }

loop:
     printf("Please select the monitoring operation:\n");
     printf("   1 : monitor locks,\n");
     printf("   2 : don't monitor locks,\n");
     printf("   3 : monitor queues,\n");
     printf("   4 : don't monitor queues,\n");
     printf("   5 : monitor transactions,\n");
     printf("   6 : don't monitor transactions,\n");
     printf("   7 : monitor deadlocks,\n");
     printf("   8 : don't monitor deadlocks,\n");
     printf("   9 : monitor connects,\n");
     printf("   10 : don't monitor connects,\n");
     printf("   11 : stop monitoring altogether.\n");
     printf("   20 : is monitoring locks?\n");
     printf("   21 : is monitoring queues?\n");
     //...
     printf("   30 : is anybody monitoring?\n");
     printf("   0 : quit this program.\n");
     scanf("%s", eventStr);
     event = atoi(eventStr);
     if (event == 0) goto done;
    try
    {
     switch (event)
         {
         case 1: mon->monitorLocks(yes); break;
         case 3: mon->monitorQueues(yes); break;
         case 5: mon->monitorTransactions(yes); break;
         case 7: mon->monitorDeadlocks(yes); break;
         case 9: mon->monitorConnects(yes); break;
         case 2: mon->monitorLocks(no); break;
         case 4: mon->monitorQueues(no); break;
         case 6: mon->monitorTransactions(no); break;
         case 8: mon->monitorDeadlocks(no); break;
```

```
            case 10: mon->monitorConnects(no); break;
            case 11: mon->stopMonitoring(); break;
            case 20: stat = mon->isMonitoringLocks();
              if (stat == yes) printf ("yes.\n");
              else printf("no.\n");
              break;
            case 21: stat = mon->isMonitoringQueues();
              if (stat == yes) printf ("yes.\n");
              else printf("no.\n");
              break;
              // ...
            case 30: stat = mon->isMonitoringActive();
              if (stat == yes) printf ("yes.\n");
              else printf("no.\n");
              break;
            case 0: break;
            default: printf("unknown request.\n");
            }
        }
        catch (oolspmException &ex)
        {
         printf ("%s.\n", ex.what());
        }
        goto loop;

    done:
         printf("End of driver.\n");
         return 0;
    }
```

B

Monitor Program Sample

The following sample program illustrates how to instantiate the following classes to monitor lock server activity and performance:

- `oolsEventsMonitor`
- `oolsEventsMonitorMessage`

This program monitors and saves the messages in a file.

Most integer constants in the header file represent a concept. In this program, these integers are printed as one or more characters which are mnemonic. These mnemonics are given in comments in the header file. For example, `oolsLockResult` of value 1 represents granting of the lock request and is printed as G.

NOTE Define the appropriate platform name before trying to compile the program.

```
#define OO_SOLARIS
//#define OO_NT

// lspm  monitor
// This program is the monitor to send the startMonitoring request
// to the lock server and to get the messages


#include "oolspm.h"



void saveUintInFile(FILE *msgFile, unsigned int ui) {
     fprintf(msgFile, "%u ", ui); }

void saveString(FILE *msgFile, const char *str) {
     fprintf(msgFile, "\"%s\" ", str);}
```

```
void saveTid(FILE *msgFile, oolsTransId tid) {
     fprintf (msgFile, "%ld ", tid); }



void saveStopCause(FILE *msgFile, oolsStopCause cause)
{
    switch (cause)
    {
    case lsKilled: fprintf(msgFile, "K "); break;
    case userStop: fprintf(msgFile, "U "); break;
    default: fprintf(msgFile, "E "); break;
    }
}

void saveLockResult(FILE *msgFile, oolsLockResult result)
{
    switch (result)
    {
    case lkerror: fprintf(msgFile, "E "); break;
    case granted: fprintf(msgFile, "G "); break;
    case upgraded: fprintf(msgFile, "U "); break;
    case queued: fprintf(msgFile, "Q "); break;
    case deadlock: fprintf(msgFile, "D "); break;
    case rejected: fprintf(msgFile, "R "); break;
    case probeSucceeded: fprintf(msgFile, "S "); break;
    case probeFailed: fprintf(msgFile, "F "); break;
    default: fprintf(msgFile, "B "); break;
    }
}

void saveResult(FILE *msgFile, oolsResult result)
{
    switch (result)
    {
    case error: fprintf(msgFile, "E "); break;
    case succeeded: fprintf(msgFile, "S "); break;
    default: fprintf(msgFile, "B "); break;
    }
}

void oolsClientId::save(FILE* msgFile)
{
    fprintf(msgFile, "%d %d \"%s\" ", uid, pid, host);
}
```

```
void saveLockType(FILE *msgFile, oolsLockType mode)
{
    switch (mode)
    {
    case noLock: fprintf(msgFile, "NL "); break;
    case IS: fprintf(msgFile, "IS "); break;
    case IC: fprintf(msgFile, "IC "); break;
    case IX: fprintf(msgFile, "IX "); break;
    case S : fprintf(msgFile, "S  "); break;
    case R : fprintf(msgFile, "R  "); break;
    case C : fprintf(msgFile, "C  "); break;
    case X : fprintf(msgFile, "X  "); break;
    default: fprintf(msgFile, "B  "); break;
    }
}

void saveETType(FILE * msgFile, oolsTransEndType type)
{
    switch (type)
    {
    case abortTran    : fprintf(msgFile, "A "); break;
    case commitTran   : fprintf(msgFile, "C "); break;
    case releaseLatchesTran: fprintf(msgFile, "R "); break;
    case downgradeTran: fprintf(msgFile, "D "); break;
    default           : fprintf(msgFile, "B "); break;
    }
}

void saveTime_t(FILE *msgFile, time_t ts) {
     fprintf(msgFile, "%d ", ts); } /* TO DO: use correct format */
#ifdef OO_NT
// nothing??
void saveSocket(FILE *msgFile, int sock) {  }
#else
void saveSocket(FILE *msgFile, int sock) {
     fprintf(msgFile, "%d ", sock); }
#endif

void oolsResourceId::save(FILE* msgFile)
{
    if (resource == FDB) fprintf(msgFile, "FD ");
    else if (resource == DB) fprintf(msgFile, "DB ");
    else fprintf(msgFile, "OC ");

    saveUintInFile(msgFile, apid);
    saveUintInFile(msgFile, fdid);
    saveUintInFile(msgFile, dbid);
```

```
    saveUintInFile(msgFile, ocid);
    saveUintInFile(msgFile, versn);
}

void oolsLockMessage::save(FILE *msgFile)
{
    saveTime_t(msgFile, ts);
    saveTid(msgFile, tid);
    resrc.save(msgFile);
    saveLockType(msgFile, mode);
    client.save(msgFile);
    saveLockResult(msgFile, result);
}

void oolsUnlockMessage::save(FILE *msgFile)
{
    saveTime_t(msgFile, ts);
    saveTid(msgFile, tid);
    resrc.save(msgFile);
    saveResult(msgFile, result);
}

void oolsBeginTransactionMessage::save(FILE *msgFile)
{
    saveTime_t(msgFile, ts);
    saveTid(msgFile, tid);
    saveUintInFile(msgFile, apid);
    saveUintInFile(msgFile, fdid);
    saveString(msgFile, fdName);
    client.save(msgFile);
    saveResult(msgFile, result);
}

void oolsEndTransactionMessage::save(FILE *msgFile)
{
    saveTime_t(msgFile, ts);
    saveTid(msgFile, tid);
    saveUintInFile(msgFile, apid);
    saveETType(msgFile, type);
    saveResult(msgFile, result);
}

void oolsDowngradeMessage::save(FILE *msgFile)
{
    saveTime_t(msgFile, ts);
    saveTid(msgFile, tid);
    saveResult(msgFile, result);
```

```
}

void oolsWaitFailedMessage::save(FILE *msgFile)
{
    saveTime_t(msgFile, ts);
    saveTid(msgFile, tid);
    resrc.save(msgFile);
    saveLockType(msgFile, mode);
    fprintf(msgFile, "%d ", uid);
}

void oolsWaitSucceededMessage::save(FILE *msgFile)
{
    saveTime_t(msgFile, ts);
    saveTid(msgFile, tid);
    resrc.save(msgFile);
    saveLockType(msgFile, mode);
    client.save(msgFile);
    fprintf(msgFile, "%d ", timeInQ);
}

void oolsDeadlockFirstMessage::save(FILE *msgFile)
{
    saveTime_t(msgFile, ts);
    saveTid(msgFile, tid);
}

void oolsDeadlockMessage::save(FILE *msgFile)
{
    saveTid(msgFile, tid);
}

void oolsDeadlockLastMessage::save(FILE *msgFile)
{
    saveTid(msgFile, tid);
}

void oolsConnectMessage::save(FILE *msgFile)
{
    saveTime_t(msgFile, ts);
    client.save(msgFile);
    saveSocket(msgFile,sock);
    saveResult(msgFile, result);
}

void oolsDisconnectMessage::save(FILE *msgFile)
{
```

```
    saveTime_t(msgFile, ts);
    saveSocket(msgFile,sock);
}

void oolsLspmStoppedMessage::save(FILE *msgFile)
{
    saveTime_t(msgFile, ts);
    saveStopCause(msgFile, cause);
}

void oolsEventsMonitorMessage::save( FILE *msgFile)
{
     /* TO DO: check for bad msgType */
    fprintf(msgFile, "%d ", msgNumber);
    fprintf(msgFile, "%d ", msgType);
    switch (msgType)
    {
     case lockMessageType:
               msg.lockMsg.save(msgFile); break;
     case unlockMessageType:
               msg.unlockMsg.save(msgFile); break;
     case beginTransactionMessageType:
               msg.btMsg.save(msgFile); break;
     case endTransactionMessageType:
               msg.etMsg.save(msgFile); break;
     case downgradeMessageType:
               msg.downgradeMsg.save(msgFile); break;
     case waitFailedMessageType:
               msg.waitFailedMsg.save(msgFile); break;
     case waitSucceededMessageType:
               msg.waitSucceededMsg.save(msgFile); break;
     case deadlockFirstMessageType:
               msg.deadlockFirstMsg.save(msgFile); break;
     case deadlockMessageType:
               msg.deadlockMsg.save(msgFile); break;
     case deadlockLastMessageType:
               msg.deadlockLastMsg.save(msgFile); break;
     case connectMessageType:
               msg.connectMsg.save(msgFile); break;
     case disconnectMessageType:
               msg.disconnectMsg.save(msgFile); break;
     case lspmStoppedMessageType:
               msg.lspmStoppedMsg.save(msgFile); break;
    };
    fprintf(msgFile, "\n");
}
```

```
FILE * msgFile;
char clientFileName[32];

int main(int argc, char* argv[])
{
     oolsEventsMonitor *mon;

     char host[MAXHOSTNAMELENGTH];
     int eventCount = 0;
     oolsEventsMonitorMessage  myMsg;

     printf("What is the host  name of the lock server to monitor?\n");
     scanf("%s", host);

     mon = new oolsEventsMonitor(host);
     /* inform lock server you are the monitor */

     try
     {
     mon->startMonitoring();
     }
     catch (oolspmLockServerNotAvailableException &ex) {
       printf("%s.\n", ex.what() );
       return 1;
       }
     catch (oolspmLockServerMonitoringAlreadyStartedException &ex) {
       printf ("%s.\n", ex.what());
       return 1;
       }

     /* get the next message and put it in a file */
     sprintf(clientFileName, "monitor%d", time(NULL));
     msgFile = fopen(clientFileName, "w");

     //int msecWait= 400000;
     int msecWait= 4000;
     char xxx[2];
     printf("[monitor.C]started monitoring.  Type any char when ready
        to get messages, wait time =%d.\n", msecWait);
     xxx[0]=getchar();
     xxx[1]=getchar();
     printf("[monitor.C]main() will save all recevied messages in a
        file.\n");

     int status = 1;
     int num_trials = 0;
```

```
     if (mon )
     {
     num_trials = 0;
     printf("time=%d\n", time(0));
     try {
loop:
       while (status = mon->getNextMessage(&myMsg, msecWait))
         {
           printf("[Monitor.C]time=%d.  Received
                    myapp.msg.msgNumber = %d,
                    msgType=%d.\n",
                    time(0),
                    myMsg.msgNumber,
                    myMsg.msgType);
           myMsg.save(msgFile);
           if ( myMsg.msgType == lspmStoppedMessageType)
              {
                 printf("Lock server says monitoring was stopped.\n");
                 break;
              }
           num_trials = 0;
         }
       if (!status)
        {
           printf("trial %d.\n", ++num_trials);
           if (num_trials < 5)
              {
              goto loop;
              }
           mon->stopMonitoring();
           printf("time=%d. Didn't receive any messages in %d msec,
                    so Stopped monitoring.\n", time(0), 5*msecWait);
        }
        }// try
        catch (oolspmLockServerConnectionClosedException &ex) {
              printf("%s.\n", ex.what());
              status = 1;
              }
     }  //if (mon)

     fclose(msgFile);
     printf("End of monitor.\n");
     return 0;
} //main
```

Index

O

R

S

T

U

W

Y

OBJECTIVITY, INC.
301B East Evelyn Avenue
Mountain View, California 94041
USA
+1 650-254-7100
+1 650-254-7171 Fax
www.objectivity.com
info@objectivity.com