11. Concurrency Management

This section first describes the concurrency management mechanisms that OLE 2 provides, and recommendations for how applications should use them. A following section gives an overview of the problems that OLE 1 applications faced in dealing with concurrency, which led to the OLE 2 design.

11.1. OLE 2 Concurrency API

11.1.1. Synchronous Method Calls

Within a single process, method calls between objects are just function calls, in all cases. When methods calls are made between objects which are in different processes, OLE 2 methods fall into three categories.

> *Asynchronous notifications,* where the caller proceeds without waiting for the callee to process the call; these calls are made through PostMessage, with no reply. Applications should not attempt to make synchronous calls while processing an asynchronous call. OLE will fail such an attempt.
>
> *Synchronous calls,* where the caller waits for the reply before continuing, and other messages may be received (in a controlled way) while waiting; these calls are made through PostMessage, with a wait loop for the reply. All methods are synchronous except those listed here as asynchronous or input-synchronized.
>
> *Input-synchronized calls*, where the call must be completed by the callee before yielding, so that focus management and type-ahead will work correctly. While processing one of these methods the callee must not yield or call any function or method (including synchronous methods) which might yield. These calls are made through Windows' SendMessage.

The following methods are asynchronous:

> IAdviseSink::OnDataChange
> IAdviseSink::OnViewChange
> IAdviseSink::OnRename
> IAdviseSink::OnSave

The following methods are input-synchronized:

> IOleWindow::GetWindow
>
> IOleInPlaceActiveObject::OnFrameWindowActivate
> IOleInPlaceActiveObject::OnDocWindowActivate
> IOleInPlaceActiveObject::ResizeBorder
>
> IOleInPlaceUIWindow::GetBorder
> IOleInPlaceUIWindow::RequestBorderSpace
> IOleInPlaceUIWindow::SetBorderSpace
>
> IOleInPlaceFrame::SetMenu
> IOleInPlaceFrame::SetStatusText
>
> IOleInPlaceObject::SetObjectRects

To minimize the problems related with asynchronous messages, OLE 2 executes the majority of object methods synchronously. Under OLE 2, applications can just call methods and not worry about dispatching and handling messages in a modal state, while waiting for the reply. When an application makes a method call, the OLE 2 libraries will enter a modal loop that handles the required message replies. This loop will (by default) permit the user to switch to other tasks, and back again, by handling the activation messages.

OLE 2 keeps track of which calls are made as a result of some other inter-process call, by assigning a logical thread ID when a top level call is made, and passing this ID around as consequent calls are made to other processes. Within its modal wait loop, OLE will identify requests that originate from the same logical thread of activity, and inform the application. The application should if at all possible accept requests that come from the same logical thread to avoid deadlocks.

A logical thread starts whenever an application starts a new OLE operation, say as a result of some user action. The logical thread that starts as a result of an incoming OLE request is not a new logical thread, and these logical threads will get the same ID as the requesting application. Thus, these operations belong to the same logical thread of execution as that of the original request. While an application is waiting for a reply, if it gets any requests from other applications with a different logical thread of execution, the request may be rejected by the application. Keeping track of the logical threads of execution and managing the thread ids is done by the OLE libraries.

The application may supply an interface that will be called while in the OLE modal loop, to optionally dispatch messages that the application can handle in this state, determine whether incoming calls can be handled, and to handle time-outs and "retry later" responses. The application specifies these functions by calling the CoRegisterMessageFilter() function described below.

11.1.1.1. CoRegisterMessageFilter

HRESULT CoRegisterMessageFilter(lpMsgFilter, lplpMsgFilterPrev)

By calling CoRegisterMessageFilter, the application can specify functions to be called in modal loops while OLE is awaiting a response to a request. CoRegisterMessageFilter() takes a pointer to an IMessageFilter interface (see below) that the application supplies. The previous value is returned through lplpMsgFilterPrev. A value of NULL indicates that the default behavior should apply, which is the case if CoRegisterMessageFilter() is not called. The default behavior is documented under the methods of the IMessageFilter interface, below.

Argument	Type	Description
lpMsgFilter	IMessageFilter*	A pointer to an IMessageFilter interface that the application supplies, or NULL
lplpMsgFilterPrev	IMessageFilter**	A pointer to a pointer where the previously installed IMessageFilter interface (or NULL) is returned
return value	HRESULT	S_OK

11.1.1.2. IMessageFilter

```
interface IMessageFilter: IUnknown {
    virtual DWORD HandleIncomingCall(dwCallType, htaskCaller, dwTickCount, dwReserved) = 0;
    virtual DWORD RetryRejectedCall(htaskCallee, dwTickCount, dwRejectType) = 0;
    virtual DWORD MessagePending(htaskCallee, dwTickcCount, dwPendingType) = 0;
    };
```

11.1.1.3. IMessageFilter::HandleIncomingCall

DWORD IMessageFilter::HandleIncomingCall(dwCallType, htaskCaller, dwTickCount, dwReserved)

This method is called when an incoming call is received, both as a result of the application calling DispatchMessage when an incoming call is in the message queue, and if one is received while attempting to make an outgoing call or awaiting a response. This method provides the application with a single point of entry for all incoming calls.

The application may use this method call to check acceptability of all method calls in its current state, and/or set up the application state in preparation for processing the call.

If the application is in a temporary state (e.g. some modal state) where it cannot process the call then it should return SERVERCALL_RETRYLATER. If it can process the message or if it might be able to, depending on which interface the call is destined for, it should return SERVERCALL_ISHANDLED. OLE will reject or process the call as indicated.

If the application returns SERVERCALL_ISHANDLED, the call may in turn fail with RPC_E_CALL_REJECTED if the application is in a state where it cannot process the particular call.

The exceptions to the above rules are input-synchronized calls and asynchronous calls, which are dispatched even if the application returns SERVERCALL_REJECTED or SERVERCALL_RETRYLATER.

Default behavior is to process the incoming call, even if it is not on the same logical thread.

Argument	Type	Description
dwCallType	DWORD	CALLTYPE indicates the kind of incoming call that has been received: whether it is a top level incoming call or nested in an outgoing call; whether it is on the same or a different logical thread, and whether the call is an asynchronous notification.
htaskCaller	HTASK	the task handle of the task which is calling this one
dwTickCount	DWORD	If dwCallType is not CALLTYPE_TOPLEVEL , this is the elapsed tick count since the outgoing call was made; otherwise this argument should be ignored
dwReserved	DWORD	reserved for future use. Applications should ignore this argument.
return value	DWORD	One of:

```
typedef enum tagCALLTYPE
    {
    CALLTYPE_TOPLEVEL = 1,   // toplevel call - no outgoing call
    CALLTYPE_NESTED = 2,     // callback on same logical thread
                             // - should always handle
    CALLTYPE_ASYNC = 3,      // asynchronous call - can NOT be rejected
    CALLTYPE_TOPLEVEL_CALLPENDING = 4,
                             // new incoming call with new logical thread
    CALLTYPE_ASYNC_CALLPENDING = 5
                             // async call from a different logical thread
                             // - can NOT be rejected
    } CALLTYPE;
```

```
typedef enum tagSERVERCALL {
    SERVERCALL_ISHANDLED = 0,
    SERVERCALL_REJECTED = 1,
    SERVERCALL_RETRYLATER = 2,
    } SERVERCALL;
```

The application should return SERVERCALL_RETRYLATER if it is in some modal state in which it cannot handle the call, and if user intervention can take the application out of the modal state, to one where it could handle the call.

Applications should only return SERVERCALL_-REJECTED if there is little or no chance that the application would later be able to handle the call, for example if some network service were not found or not accessible.

Handling of incoming calls is illustrated in Figure 96.

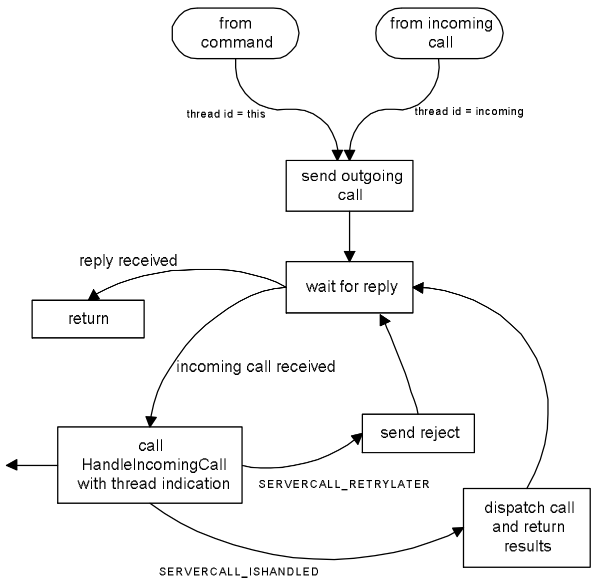

Figure 96.

11.1.1.4. IMessageFilter::RetryRejectedCall

DWORD IMessageFilter::RetryRejectedCall(htaskCallee, dwTickCount, dwRejectType)

If the server task rejects a call, OLE will check its state (disconnected etc.) and check to see if there are incoming calls (which might be from the server that this task tried to call). If there are, OLE processes the calls (after checking with HandleIncomingCall described above) and retries. If there are not, the implication is that the server task is in some state where it cannot handle such calls, perhaps temporarily. In this case, OLE calls RetryRejectedCall, to give the application the chance to put up a dialog giving the user the choice to retry, cancel the call, or switch to the task identified by hTaskCallee, and let the user decide whether to retry the call, or cancel. If the user elects to cancel the action, the call will appear to fail with RPC_E_CALL_REJECTED.

Applications should only retry calls that returned SERVERCALL_REJECTED in very special cases, where they have specific knowledge about the state of the server.

Default behavior is to fail the call with RPC_E_CALL_REJECTED.

Argument	Type	Description
hTaskCallee	HTASK	the task handle of the server task that rejected the call
dwTickCount	DWORD	the elapsed ticks since the call was made
dwRejectType	DWORD	one of SERVERCALL_REJECTED or SERVERCALL_RETRYLATER, as returned by the server application.
return value	DWORD	-1 if the call should be canceled, in which case OLE will return RPC_E_CALL_CANCELLED from the original method call
		Between 0 and 100: the call will be retried immediately
		Greater than 100: OLE will wait for this many milliseconds and then retry.
		Normally, applications should either retry immediately (after the user has made that choice) or cancel. The option to wait and retry is provided for special kinds of calling application such as background tasks executing macros or scripts, so that they can retry in a non-intrusive way.

Handling of rejected calls is illustrated in Figure 97.

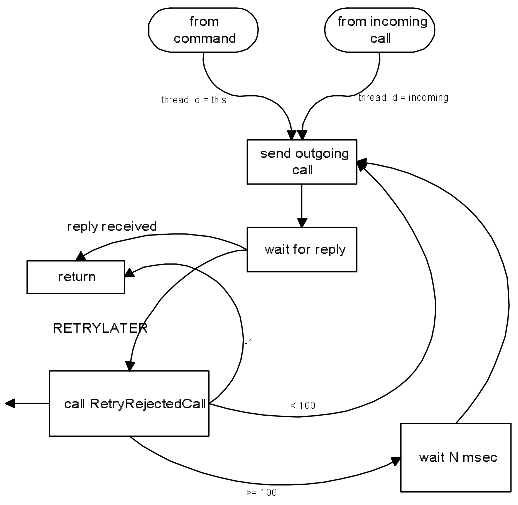

Figure 97.

11.1.1.5. IMessageFilter::MessagePending

DWORD MessageFilter::MessagePending(htaskCallee, dwTickCount, dwPendingType)

Handling input while waiting for an outgoing call to complete introduces some complications. If the callee application processes input while executing the method call (e.g. in a dialog), then it will normally have taken focus before the caller sees any input, but not if it does not take focus immediately. In this situation, the user might switch back to the caller application and try to work with it.

If the callee application does not take focus, and will complete the operation after some period of time (in particular without user intervention) then user input will appear in the caller application's message queue, and in the particular case of type-ahead by frequent or expert users, the user will expect that input to be processed on completion of the method call, just as it would if the object were not in a separate process.

When input appears in the queue, which might either be type-ahead or the user trying to get attention from the application, the caller application should leave messages in the queue (perhaps dispatching

WM_PAINT and WM_MOUSEMOVE messages) until sufficient time has passed that the user's input is probably not type-ahead but is an attempt to get attention; somewhere between 2 and 3 seconds is recommended. If that amount of time has passed and the call has not completed (which the application determines when it receives a call to MessagePending with appropriate dwTickCount), the application should flush input messages from the queue, and show a dialog offering the user the choice to retry (i.e. keep waiting) or switch to the task identified by hTaskCallee.

The above rule ensures that if calls do complete in a reasonable time, type-ahead will be treated correctly, but if the callee does not respond, the type-ahead is not misinterpreted, and the user is able to take action to rectify the problem. An example of the problem is OLE 1.0 servers which queue up requests, without responding, when they are in modal dialogs etc.

If the application has installed a message filter with CoRegisterMessageFilter, and a Windows message appears in the queue while OLE is awaiting a reply to a remote call, OLE will determine what it can about the state of the server. If the connection is lost or the server has terminated, the original call will fail with RPC_E_CONNECTION_TERMINATED or RPC_E_SERVER_DIED. If there has been no response, OLE will calculate the elapsed time since the call was made, and call MessagePending. OLE will not remove the message from the queue. The application should determine whether to process the message without interrupting the call, continue waiting, or cancel the operation. The application can offer a dialog, and/or switch to the task indicated by hTaskCallee, before returning from the call. If the application returns PENDINGMSG_CANCELCALL, OLE will fail the original call and return RPC_E_CALL_CANCELLED.

In the case of a server which does not respond, the application can cancel the call, and recover the OLE object to a consistent state by Revert() on its storage. The object can simply be released if and when the client application wants to shut down. However, canceling a call will leave orphaned operations and will leak resources. Canceling should only be used as a last resort, and applications are strongly recommended not to allow it at all.

Default behavior if there is no message filter installed is to behave as if MessagePending had returned PENDINGMSG_WAITDEFPROCESS - that is, task switching (alt-tab etc.) and window activation messages are dispatched, WM_PAINT, WM_TIMER are dispatched, other input messages are discarded, and OLE continues to wait for the reply.

Argument	Type	Description
hTaskCallee	HTASK	the task handle of the server application to which a call is being made and which has not yet replied
dwTickCount	DWORD	the elapsed time (calculated from GetTickCount()) since the call was made.
dwPendingType	DWORD	indicates the type of call during which some Windows message was received, one of: `typedef enum tagPENDINGTYPE {` `PENDINGTYPE_TOPLEVEL = 1, // toplevel call` `PENDINGTYPE_NESTED = 2, // nested call` `} PENDINGTYPE;`
return value	DWORD	one of: `typedef enum tagPENDINGMSG {` `PENDINGMSG_CANCELCALL = 0, // cancel the outgoing call` `PENDINGMSG_WAITNOPROCESS = 1, // continue waiting for the reply and don't dispatch the message` `PENDINGMSG_WAITDEFPROCESS = 2 // dispatch the message and then continue waiting` `} PENDINGMSG;` PENDINGMSG_CANCELCALL should only be returned under extreme duress. Canceling a call that has not replied (or been rejected) will create orphan transactions and will lose resources. PENDINGMSG_WAITNOPROCESS causes OLE to leave the message in the queue, and continue waiting. A subsequent message will trigger

another call to MessagePending. Leaving messages in the queue enables them to be processed normally if the outgoing call completes.

PENDINGMSG_WAITDEFPROCESS invokes the OLE default message handling, which is to dispatch WM_PAINT, WM_TIMER, WM_MOUSEMOVE, all posted messages, and to discard input messages other than task switching messages.

If a PENDINGTYPE other than one of the above enumerated values is received, applications should return PENDINGMSG_WAITDEFPROCESS. This allows future RPC implementations to inform the application of other notifications (such as call completed while the callback was showing a dialog).

Handling of pending messages is illustrated in Figure 98.

11.1.2. Blocking while busy

If an application dispatches messages while it is executing a command or an object method (as a result of an incoming call), the OLE libraries will call HandleIncomingCall as explained above. If the application can not handle calls during this time, it should return SERVERCALL_RETRYLATER. The application is responsible for tracking its state so it can respond to HandleIncomingCall appropriately.

These functions should not be used to hold off updates to objects during operations like band-printing. Instead, IViewObject::Freeze() should be used for that purpose.

Note that asynchronous notifications cannot be rejected. The application must always be able to process these notifications.

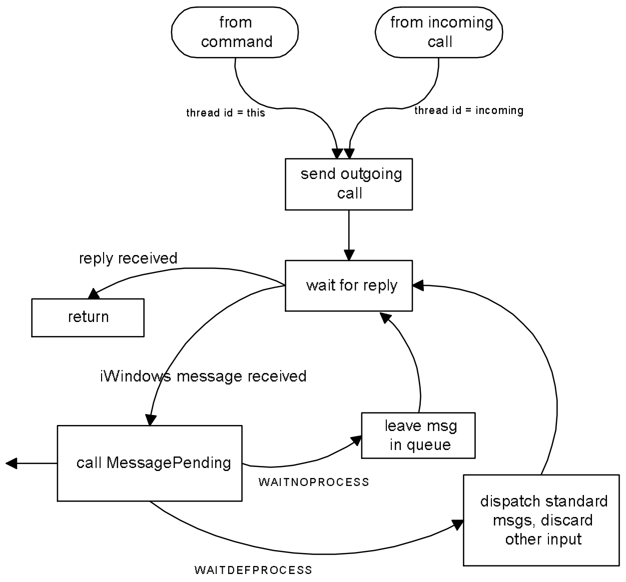

Figure 98.

11.2. Background: OLE 1 Concurrency Problems

OLE 1 introduced problems for developers of client applications in the form of functions that do not complete immediately, but do so at a later time, including freeing the object when finished with it. Server applications were faced with complex rules about the order in which callbacks are made, and how to block out requests while busy.

Additionally, OLE 1 suffered from internal problems arising from the underlying DDE protocol, which resulted in less than optimal support for applications, for example in error recovery.

11.2.1. Client Applications

Superficially, OLE 1 provides client applications with a relatively simple functional interface to its objects. In reality, programming a client is more complex.

11.2.1.1. Asynchronous Operations

Many of the OLE 1 function calls trigger operations on objects involving the server, which are inherently asynchronous because servers communicate between different applications. Rather than making the operations appear synchronous to the client, OLE 1 explicitly requires the client to be aware of and to manage the asynchrony. OLE does this by returning a result code indicating that the operation is not yet complete. The client is expected to handle this situation and treat the object as busy until OLE notifies completion (or failure) of the operation.

11.2.1.2. Sequences of Operations

The difficulty of managing the asynchrony is compounded if the client wants to perform a series of operations on the object. One occurrence of this is when activating an object for editing, the client may want to send a series of messages setting ambient state information in the server. OLE 1 requires the client to wait for one operation to complete before invoking another. This puts the burden on the client of handling the wait loop, dispatching messages so that the reply can be processed, and deciding what to do with other messages that may come in (e.g., user input). It is hard for the client to return to its normal dispatch loop for this purpose, since that would require that the client manage the sequencing via a state machine of some kind, which is extra programming effort. We have provided sample code that shows a prototypical wait loop, which permits the user to switch away from the application but otherwise waits for the operation to complete before proceeding. However, this code is tricky to integrate into an application, and has caused considerable trouble for developers.

11.2.1.3. Shutdown Operations

The inherent weakness in the underlying messaging mechanism is reflected out to the client applications in the form of complicated rules governing termination of the application. In essence, the application is required to keep track of the set of objects that have not shut down properly, and wait around until they do so. Ideally applications should give the user the impression that they have closed by hiding windows etc., and they should free as many resources as they can before entering this wait loop. In practice this can be a difficult thing to do given the way some applications are structured.

11.2.1.4. Blocking Objects

OLE 1 relies on the client's message dispatch mechanism to manage its communication. OLE makes the assumption that if the client dispatched the message, then it is OK to process it. This puts the burden on the client of deciding which messages to dispatch immediately and which ones to queue if they cannot be processed, etc. Further, OLE gives no help in determining what messages should be dispatched.

An example of requirement for blocking is to support printing in bands, where the client wishes to check for user input (e.g. to cancel printing) but not let the object change state during the printing.

11.2.1.5. What Can be Done in Callbacks?

This is largely a documentation problem. Callbacks are calls to methods in interfaces that applications implement. These calls may occur either during a call that the client makes to OLE, or as a result of dispatching a message. In OLE 1 the application should not dispatch messages or make calls to objects in other processes during the callback, as this would cause re-entrancy problems both in the application and in the OLE implementation.

11.2.1.6. Knowing the Server's State

OLE 1 uses a communication mechanism that makes it impossible to determine whether the server is busy, unable to respond to requests, in a modal state, and so on. Further, there is no way to pass back adequate error information to clients.

11.2.1.7. Defensive Coding

It is a general rule with Windows applications that developers do not want the integrity of their application (and hence the user's data) to depend on good behavior on the part of other applications. Although to some extent Windows itself imposes this dependency, developers generally want to be able to handle cases of the other application crashing, simply never responding, or arbitrarily long gaps in processing, without locking out the user. Usually this means some kind of time-out, retry or cancel, and cleanup code.

11.2.2. Server Applications

Servers have the problem of being able to respond to requests both from the user and from clients. They need to ensure that these do not conflict, by blocking out client requests while they are busy on behalf of the user or another client. In addition, the server application often has to be able to run either as a slave to a client (e.g. during invisible update by a client), or as a stand-alone application which also serves clients.

11.2.2.1. Blocking: Objects, Documents, Servers

OLE 1 permits the server to block out messages which causes OLE to queue the messages internally until the server un-blocks. This permits the server to continue to dispatch messages without fear of getting an intrusive callback from OLE.

One point of confusion is that the OLE 1 un-block call only processes one message before returning; if the server wishes to un-block them all it must call the un-block function in a loop until it returns "all done". The intent of this is to avoid complications if processing one message requires a further block, and to permit servers to control the processing of queued messages, possibly interleaving other actions (such as peeking for user input).

Another possible problem is that we do not provide for finer granularity blocking, at the object or document level. At present, there is no plan to provide finer granularity of blocking.

11.2.2.2. Startup Rules

OLE 1 Rules for how servers should behave at startup, how to process command lines, whether to show windows etc. are complicated and unclear. Further complication arose from the introduction of verbs in addition to the original "activate for editing" functionality.

11.2.2.3. Shutdown Rules

OLE 1 Servers face the same problems as clients concerning shutdown, with the further complication of when to actually exit. This is particular troublesome for applications that act as servers for multiple object types.

11.2.3. Internal Problems

11.2.3.1. Initiation

In OLE 1, getting connected to the right instance of the right application and talking to the right occurrence of a loaded document was a matter of luck at best.

11.2.3.2. Two-message update

Since objects were represented by both native and presentation data formats, and these were updated separately in the communication channel, it was possible to have inconsistent objects, and ambiguity about when to notify the client.

11.2.3.3. Termination

OLE 1 cannot provide clean termination of conversations without the client's assistance.

11.2.3.4. Error Notification

As noted earlier, there was no way to get a proper error indication back to the client.