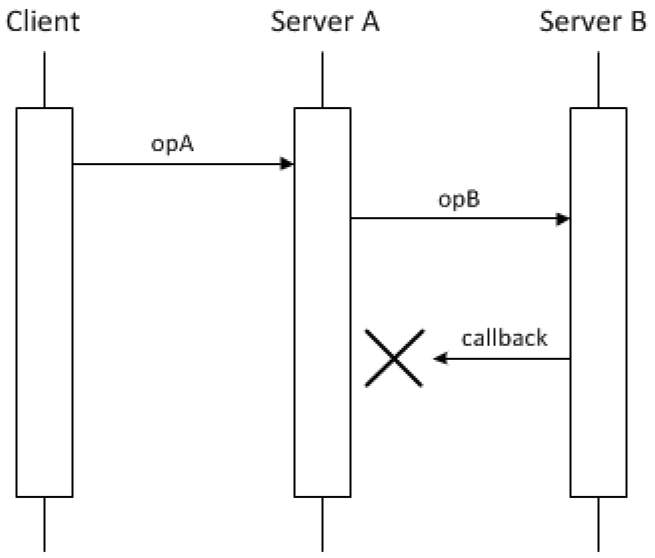
Nested Invocations

A *nested invocation* is one that is made within the context of another Ice operation. For instance, the implementation of an operation in a servant might need to make a nested invocation on some other object, or an AMI callback object might invoke an operation in the course of processing a reply to an asynchronous request. It is also possible for one of these invocations to result in a nested callback to the originating process. The maximum depth of such invocations is determined by the size of the thread pools used by the communicating parties.

On this page:

Deadlocks with Nested Invocations

Applications that use nested invocations must be carefully designed to avoid the potential for deadlock, which can easily occur when invocations take a circular path. For example, this illustration presents a deadlock scenario when using the default thread pool configuration:



Nested invocation deadlock.

In this diagram, the implementation of opA makes a nested twoway invocation of opB, but the implementation of opB causes a deadlock when it tries to make a nested callback. As mentioned in Thread Pools, the communicator's thread pools have a maximum size of one thread unless explicitly configured otherwise. In Server A, the only thread in the server thread pool is busy waiting for its invocation of opB to complete, and therefore no threads remain to handle the callback from Server B. The client is now blocked because Server A is blocked, and they remain blocked indefinitely unless timeouts are used.

There are several ways to avoid a deadlock in this scenario:

- **Increase the maximum size of the server thread pool in Server A.**

  Configuring the server thread pool in Server A to support more than one thread allows the nested callback to proceed. This is the simplest solution, but it requires that you know in advance how deeply nested the invocations may occur, or that you set the maximum size to a sufficiently large value that exhausting the pool becomes unlikely. For example, setting the maximum size to two avoids a deadlock when a single client is involved, but a deadlock could easily occur again if multiple clients invoke opA simultaneously. Furthermore, setting the maximum size too large can cause its own set of problems.

- **Use a oneway invocation.**

  If Server A called opB using a oneway invocation, it would no longer need to wait for a response and therefore opA could complete, making a thread available to handle the callback from Server B. However, we have made a significant change in the semantics of opA because now there is no guarantee that opB has completed before opA returns, and it is still possible for the oneway invocation of opB to block.

- **Create another object adapter for the callbacks.**

  No deadlock occurs if the callback from Server B is directed to a different object adapter that is configured with its own thread pool.

- **Implement opA using asynchronous dispatch and invocation.**

By declaring `opA` as an AMD operation and invoking `opB` using AMI, Server A can avoid blocking the thread pool's thread while it waits for `opB` to complete. This technique, known as *asynchronous request chaining*, is used extensively in Ice services such as IceGrid and Glacier2 to eliminate the possibility of deadlocks.

As another example, consider a client that makes a nested invocation from an AMI callback object using the default thread pool configuration. The (one and only) thread in the client thread pool receives the reply to the asynchronous request and invokes its callback object. If the callback object in turn makes a nested twoway invocation, a deadlock occurs because no more threads are available in the client thread pool to process the reply to the nested invocation. The solutions are similar to some of those presented in the above illustration: increase the maximum size of the client thread pool, use a oneway invocation, or call the nested invocation using AMI.

Analyzing an Application for Nested Invocations

A number of factors must be considered when evaluating whether an application is properly designed and configured for nested invocations:

- The thread pool configurations in use by all communicating parties have a significant impact on an application's ability to use nested invocations. While analyzing the path of circular invocations, you must pay careful attention to the threads involved to determine whether sufficient threads are available to avoid deadlock. This includes not just the threads that dispatch requests, but also the threads that make the requests and process the replies. Enabling the `Ice.Trace.ThreadPool` property can give you a better understanding of the thread pool behavior in your application.
- Bidirectional connections are another complication, since you must be aware of which threads are used on either end of the connection.
- Finally, the synchronization activities of the communicating parties must also be scrutinized. For example, a deadlock is much more likely when a lock is held while making an invocation.

As you can imagine, tracing the call flow of a distributed application to ensure there is no possibility of deadlock can quickly become a complex and tedious process. In general, it is best to avoid circular invocations if at all possible.

See Also

- Thread Pools
- Object Adapter Thread Pools
- Thread Pool Design Considerations
- Oneway Invocations