Thread Pool Design Considerations

Improper configuration of a thread pool can have a serious impact on the performance of your application. This page discusses some issues that you should consider when designing and configuring your applications.

On this page:

Single-Threaded Pool

There are several implications of using a thread pool with a maximum size of one thread:

- **Only one message can be dispatched at a time.**

  This can be convenient because it lets you avoid (or postpone) dealing with thread-safety issues in your application. However, it also eliminates the possibility of dispatching requests concurrently, which can be a bottleneck for applications running on multi-CPU systems or that perform blocking operations. Another option is to enable serialization in a multi-threaded pool.

- **Only one AMI reply can be processed at a time.**

  An application must increase the size of the client thread pool in order to process multiple AMI callbacks in parallel.

- **Nested twoway invocations are limited.**

  At most one level of nested twoway invocations is possible.

It is important to remember that a communicator's client and server thread pools have a default maximum size of one thread, therefore these limitations also apply to any object adapter that shares the communicator's thread pools.

Back to Top ^

Multi-Threaded Pool

Configuring a thread pool to support multiple threads implies that the application is prepared for the Ice run time to dispatch operation invocations or AMI callbacks concurrently. Although greater effort is required to design a thread-safe application, you are rewarded with the ability to improve the application's scalability and throughput.

Choosing an appropriate maximum size for a thread pool requires careful analysis of your application. For example, in compute-bound applications it is best to limit the number of threads to the number of physical processor cores or threads on the host machine; adding any more threads only increases context switches and reduces performance. Increasing the size of the pool beyond the number of cores can improve responsiveness when threads can become blocked while waiting for the operating system to complete a task, such as a network or file operation. On the other hand, a thread pool configured with too many threads can have the opposite effect and negatively impact performance. Testing your application in a realistic environment is the recommended way of determining the optimum size for a thread pool.

If your application uses nested invocations, it is very important that you evaluate whether it is possible for thread starvation to cause a deadlock. Increasing the size of a thread pool can lessen the chance of a deadlock, but other design solutions are usually preferred.

Back to Top ^

Serializing Requests in a Multi-Threaded Pool

When using a multi-threaded pool, the nondeterministic nature of thread scheduling means that requests from the same connection may not be dispatched in the order they were received. Some applications cannot tolerate this behavior, such as a transaction processing server that must guarantee that requests are executed in order. There are two ways of satisfying this requirement:

1. Use a single-threaded pool.
2. Configure a multi-threaded pool to serialize requests using its `Serialize` property.

At first glance these two options may seem equivalent, but there is a significant difference: a single-threaded pool can only dispatch one request at a time and therefore serializes requests from *all* connections, whereas a multi-threaded pool configured for serialization can dispatch requests from different connections concurrently while serializing requests from the same connection.

For requests dispatched using asynchronous method dispatch (AMD), `Serialize` only serializes the dispatching, not the requests themselves. Ice will dispatch the next request once its dispatch its complete–it does not wait for the first request to provide a response or exception.

You can obtain a comparable behavior from a multi-threaded pool without enabling serialization, but only if you design the clients so that they do not send requests from multiple threads, do not send requests over more than one connection, and only use synchronous twoway invocations. In general, however, it is better to avoid such tight coupling between the implementations of the client and server.

Enabling serialization can improve responsiveness and performance compared to a single-threaded pool, but there is an associated cost. The extra synchronization that the pool must perform to serialize requests can add significant overhead and result in higher latency and reduced throughput.

As you can see, thread pool serialization is not a feature that you should enable without analyzing whether the benefits are worthwhile. For example, it might be an inappropriate choice for a server with long-running operations when the client needs the ability to have several operations in progress simultaneously. If serialization was enabled in this situation, the client would be forced to work around it by opening several connections to the server, which again tightly couples the client and server implementations. If the server must keep track of the order of client requests, a better solution would be to use serialization in conjunction with AMD to queue the incoming requests for execution by other threads.

Back to Top ^

See Also

- Thread Pools
- Concurrent Proxy Invocations
- Nested Invocations
- Thread Safety
- Connection Establishment

Previous

Next