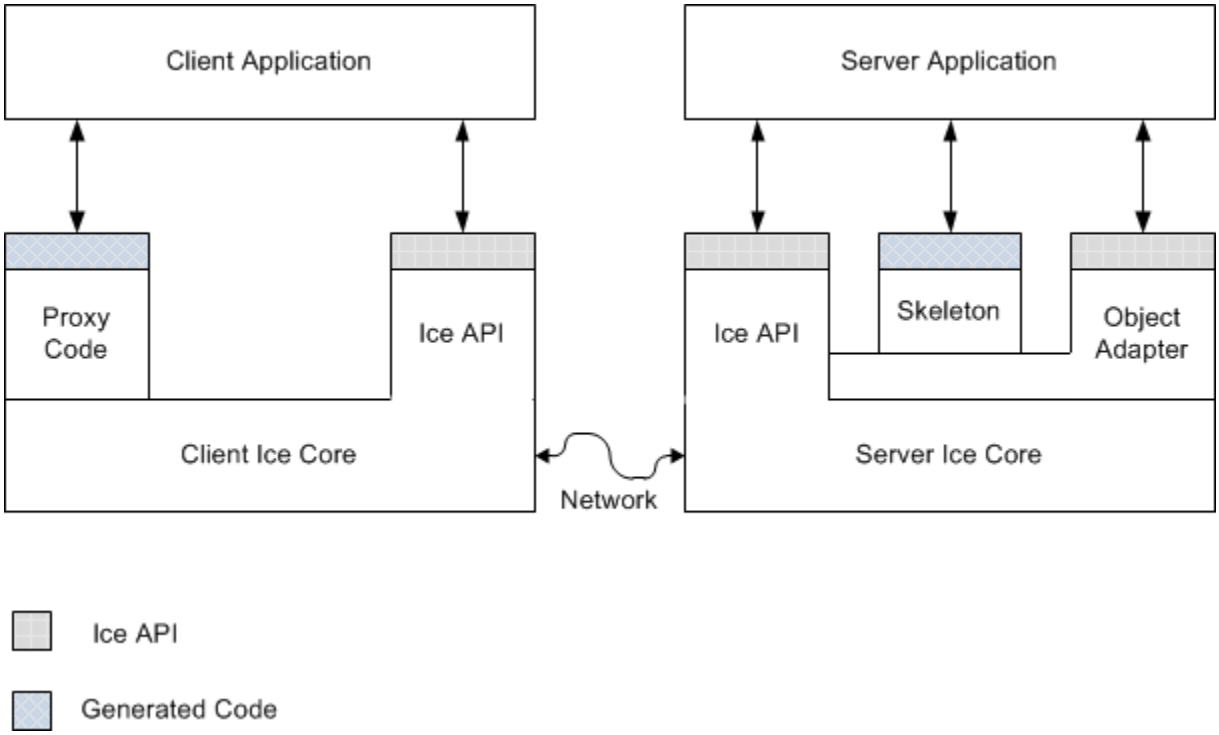
Client and Server Structure

Ice clients and servers have the logical internal structure:



Ice Client and Server Structure

Both client and server consist of a mixture of application code, library code, and code generated from Slice definitions:

- The Ice core contains the client- and server-side run-time support for remote communication. Much of this code is concerned with the details of networking, threading, byte ordering, and many other networking-related issues that we want to keep away from application code. The Ice core is provided as a number of libraries that client and server use.
- The generic part of the Ice core (that is, the part that is independent of the specific types you have defined in Slice) is accessed through the Ice API. You use the Ice API to take care of administrative chores, such as initializing and finalizing the Ice run time. The Ice API is identical for clients and servers (although servers use a larger part of the API than clients).
- The proxy code is generated from your Slice definitions and, therefore, specific to the types of objects and data you have defined in Slice. The proxy code has two major functions:
  - It provides a down-call interface for the client. Calling a function in the generated proxy API ultimately ends up sending an RPC message to the server that invokes a corresponding function on the target object.
  - It provides *marshaling* and *unmarshaling* code. Marshaling is the process of serializing a complex data structure, such as a sequence or a dictionary, for transmission on the wire. The marshaling code converts data into a form that is standardized for transmission and independent of the endian-ness and padding rules of the local machine. Unmarshaling is the reverse of marshaling, that is, deserializing data that arrives over the network and reconstructing a local representation of the data in types that are appropriate for the programming language in use.
- The skeleton code is also generated from your Slice definition and, therefore, specific to the types of objects and data you have defined in Slice. The skeleton code is the server-side equivalent of the client-side proxy code: it provides an up-call interface that permits the Ice run time to transfer the thread of control to the application code you write. The skeleton also contains marshaling and unmarshaling code, so the server can receive parameters sent by the client, and return parameters and exceptions to the client.
- The object adapter is a part of the Ice API that is specific to the server side: only servers use object adapters. An object adapter has several functions:
  - The object adapter maps incoming requests from clients to specific methods on programming-language objects. In other words, the object adapter tracks which servants with what object identity are in memory.
  - The object adapter is associated with one or more transport endpoints. If more than one transport endpoint is associated with an adapter, the servants incarnating objects within the adapter can be reached via multiple transports. For example, you can associate both a TCP/IP and a UDP endpoint with an adapter, to provide alternate quality-of-service and performance characteristics.
  - The object adapter is responsible for the creation of proxies that can be passed to clients. The object adapter knows about the type, identity, and transport details of each of its objects and embeds the correct details when the server-side application code requests the creation of a proxy.

Note that, as far as the process view is concerned, there are only two processes involved: the client and the server. All the run time support for distributed communication is provided by the Ice libraries and the code that is generated from Slice definitions. (For indirect proxies, a third process, IceGrid, is required to resolve proxies to transport endpoints.)

See Also

- Hello World Application
- IceGrid