Communicator Initialization

Previous Next

On this page:

- Creating a Communicator with initialize
- InitializationData

Creating a Communicator with `initialize`

The first step for any Ice application is to initialize the Ice run time by calling the Ice `initialize` native function or method. (`initialize` is named `createCommunicator` in Objective-C). `initialize` returns a `Communicators` object, which represents an instance of the Ice run time.

You can either call `initialize` directly, or you can use a helper class that calls `initialize` under the hood, such as `Application` (available in most language mappings) or `Ice::CommunicatorHolder` in C++.

Each language mapping provides several overloads for `initialize` with the following parameters:

1. the arguments given to the application
   This `initialize` extracts Ice properties from the provided arguments, and sets them in the newly created communicator. See Command-Line Parsing and Initialization for further details.

2. the arguments given to the application plus an `InitializationData` parameter
   This `initialize` uses the `InitializationData` parameter (described below on this page) to create a new communicator, and then extracts Ice properties from the provided arguments. The properties extracted from the arguments overwrite properties set in `InitializationData`.

3. the arguments given to the application plus a config file parameter
   This `initialize` is a shortcut for the `initialize` in 2. above, where the `InitializationData` parameter contains only properties loaded from the provided configuration file. For example, here is the corresponding implementation in Java:

   Java

   ```
   public static Communicator initialize(String[] args, String configFile, java.util.List<String>
   remainingArgs)
   {
       InitializationData initData = null;
       if(configFile != null)
       {
           initData = new InitializationData();
           initData.properties = Util.createProperties();
           initData.properties.load(configFile);
       }
       return initialize(args, initData, remainingArgs);
   }
   ```

4. an `InitializationData` parameter
   This `initialize` uses the `InitializationData` parameter (described below on this page) to create a new communicator. It's similar to `initialize` in 2. above, except this `initialize` does not take arguments and does not check the `ICE_CONFIG` environment variable.

5. a config file parameter
   This `initialize` is a shortcut for the `initialize` in 4. above, where the `InitializationData` parameter contains only properties loaded from the provided configuration file.

6. no parameters at all
   This `initialize` creates a communicator with no properties (meaning all Ice properties have their default value), and only default features. It also does not check the `ICE_CONFIG` environment variable.

C++11

```
namespace Ice
{
    // (1) and (2): argc/argv + InitializationData
    std::shared<Communicator> initialize(int& argv, const char* argv[], const InitializationData& initData =
InitializationData(),
                                         int = ICE_INT_VERSION);

    // (3): argc/argv + config file
    std::shared<Communicator> initialize(int& argc, const char* argv[], const std::string& configFile,
                                         int version = ICE_INT_VERSION);

#ifdef _WIN32
    // (1) and (2): wide argc/argv + InitializationData
    std::shared<Communicator> initialize(int& argc, const wchar_t* argv[], const InitializationData& initData =
InitializationData(),
                                         int version = ICE_INT_VERSION);

    // (3): wide argc/argv + config file
    std::shared<Communicator> initialize(int& argc, const wchar_t* argv[], const std::string& configFile,
                                         int version = ICE_INT_VERSION);
#endif

    // (1) and (2): args as a vector<string> + InitializationData
    std::shared<Communicator> initialize(StringSeq& args, const InitializationData& initData =
InitializationData(),
                                         int version = ICE_INT_VERSION);

    // (3): args as a vector<string> + config file
    std::shared<Communicator> initialize(StringSeq& args, const std::string& configFile,
                                         int version = ICE_INT_VERSION);

    // (4) and (6): no args + InitializationData
    std::shared<Communicator> initialize(const InitializationData& initData = InitializationData(),
                                         int version = ICE_INT_VERSION);

    // (5) no args + config file
    std::shared<Communicator> initialize(const std::string& configFile,
                                         int version= ICE_INT_VERSION);
}
```

All `initialize` functions have a trailing parameter `int version = ICE_INT_VERSION`. This parameter ensures the Ice version in the header files you used to compile your application is compatible with the Ice version in the Ice libraries loaded by this application. You should always keep the default value (`ICE_INT_VERSION`) for this parameter.

C++98

```
namespace Ice
{
    // (1) and (2): argc/argv + InitializationData
    CommunicatorPtr initialize(int& argv, const char* argv[], const InitializationData& initData =
InitializationData(),
                               int = ICE_INT_VERSION);

    // (3): argc/argv + config file
    CommunicatorPtr initialize(int& argc, const char* argv[], const char* configFile,
                               int version = ICE_INT_VERSION);

#ifdef _WIN32
    // (1) and (2): wide argc/argv + InitializationData
    CommunicatorPtr initialize(int& argc, const wchar_t* argv[], const InitializationData& initData =
InitializationData(),
                               int version = ICE_INT_VERSION);

    // (3): wide argc/argv + config file
    CommunicatorPtr initialize(int& argc, const wchar_t* argv[], const char* configFile,
                               int version = ICE_INT_VERSION);
#endif

    // (1) and (2): args as a vector<string> + InitializationData
    CommunicatorPtr initialize(StringSeq& args, const InitializationData& initData = InitializationData(),
                               int version = ICE_INT_VERSION);

    // (3): args as a vector<string> + config file
    CommunicatorPtr initialize(StringSeq& args, const char* configFile,
                               int version = ICE_INT_VERSION);

    // (4) and (6): no args + optional InitializationData
    CommunicatorPtr initialize(const InitializationData& initData = InitializationData(),
                               int version = ICE_INT_VERSION);

    // (5): no args + config file
    CommunicatorPtr initialize(const char* configFile,
                               int version= ICE_INT_VERSION);
}
```

All `initialize` functions have a trailing parameter `int version = ICE_INT_VERSION`. This parameter ensures the Ice version in the header files you used to compile your application is compatible with the Ice version in the Ice libraries loaded by this application. You should always keep the default value (`ICE_INT_VERSION`) for this parameter.

C#

```
namespace Ice
{
    public sealed class Util
    {
        // (1): args only
        public static Communicator initialize(ref string[] args) { ... }

        // (2): args + InitializationData
        public static Communicator initialize(ref string[] args, InitializationData initData) { ... }

        // (3): args + config file
        public static Communicator initialize(ref string[] args, string configFile) { ... }

        // (4): no args + InitializationData
        public static Communicator initialize(InitializationData initData) { ... }

        // (5): no args + config file
        public static Communicator initialize(string configFile) { ... }

        // (6): no parameter at all
        public static Communicator initialize() { ... }

        ...
     }
}
```

Java

```
package com.zeroc.Ice;

public final class Util
{
    // (1): args only
    public static Communicator initialize(String[] args) { ... }
    public static Communicator initialize(String[] args, java.util.List<String> remainingArgs) { ... }

    // (2): args + InitializationData
    public static Communicator initialize(String[] args, InitializationData initData) { ... }
    public static Communicator initialize(String[] args, InitializationData initData, java.util.List<String>
remainingArgs) { ... }

    // (3): args + config file
    public static Communicator initialize(String[] args, String configFile) { ... }
    public static Communicator initialize(String[] args, String configFile, java.util.List<String>
remainingArgs) { ... }

    // (4): no args + InitializationData
    public static Communicator initialize(InitializationData initData) { ... }

    // (5): no args + config file
    public static Communicator initialize(String configFile) { ... }

    // (6): no parameter at all
    public static Communicator initialize() { ... }
    ...
}
```

Java Compat

```
package Ice;

public final class Util
{
    // (1): args only
    public static Communicator initialize(String[] args) { ... }
    public static Communicator initialize(StringSeqHolder args) { ... }

    // (2): args + InitializationData
    public static Communicator initialize(String[] args, InitializationData initData) { ... }
    public static Communicator initialize(StringSeqHolder args, InitializationData initData) { ... }

    // (3): args + config file
    public static Communicator initialize(String[] args, String configFile) { ... }
    public static Communicator initialize(StringSeqHolder args, String configFile) { ... }

    // (4): no args + InitializationData
    public static Communicator initialize(InitializationData initData) { ... }

    // (5) no args + config file
    public static Communicator initialize(String configFile) { ... }

    // (6): no parameter at all
    public static Communicator initialize() { ... }
    ...
}
```

JavaScript

```
// (1): args
// (2): args + InitializationData
// (4): InitializationData
// (6): no parameter at all
Ice.initialize = function(arg1, arg2)
```

MATLAB

```
% in Ice package

% initialize returns the new communicator and the remaining args

% (1): args (a cell array of character vectors)
% (2): args + InitializationData
% (3): args + config file
% (4): InitializationData
% (5): config file
% (6): no parameter at all
function [communicator, remainingArgs] = initialize(varargin)
```

ObjC

```
@interface ICEUtil : NSObject
// (1): argc/argv only
+(id<ICECommunicator>) createCommunicator:(int*)argc argv:(char*[])argv;

// (2): argc/argv + ICEInitializationData
+(id<ICECommunicator>) createCommunicator:(int*)argc argv:(char*[])argv initData:(ICEInitializationData*)
initData;

// (3): argc/argv + config file
+(id<ICECommunicator>) createCommunicator:(int*)argc argv:(char*[])argv configFile:(NSString*)configFile;

// (4): no args + InitializationData
+(id<ICECommunicator>) createCommunicator:(ICEInitializationData*)initData;

// (5) not provided in Objective-C due to overload ambiguity
// (6): no parameter at all
+(id<ICECommunicator>) createCommunicator;
...
@end
```

PHP

```
namespace Ice
{
    // (1): args
    // (2): args + InitializationData
    // (4): InitializationData
    // (6): no parameter at all
    function initialize(args=null, initData=null)
}
```

Python

```
# in Ice module

# (1): args
# (2): args + InitializationData
# (3): args + config file
# (4): InitializationData
# (5): config file
# (6): no parameter at all
def initialize(args=None, data=None)
```

Ruby

```
module Ice
    # (1): args
    # (2): args + InitializationData
    # (3): args + config file
    # (4): InitializationData
    # (5): config file
    # (6): no parameter at all
    def Ice.initialize(args=nil, initData=nil)
```

Back to Top ^

InitializationData

During the creation of a communicator, the Ice run time initializes a number of features that affect the communicator's operation. Once set, these features remain in effect for the life time of the communicator, that is, you cannot change these features after you have created a communicator. Therefore, if you want to customize these features, you must do so when you create the communicator.

The following features can be customized at communicator creation time:

- the property set
- the logger object
- the instrumentation observer
- the thread start and stop notification hooks
- the dispatcher
- the compact ID resolver (used by the streaming interfaces when extracting Ice objects)
- the class loader (Java only)
- the batch request interceptor
- the value factory manager

To establish these features, you initialize a structure or class of type `InitializationData`, set one or more fields of this structure or class, and pass this `InitializationData` to `initialize` or to a helper class (such as Ice `Application`) that calls `initialize` for you.

The exact definition of `InitializationData` depends on the language mapping you use, and some language mappings only support a subset of these features:

C++11

```
namespace Ice
{
    struct InitializationData
    {
        std::shared_ptr<Ice::Properties> properties;
        std::shared_ptr<Ice::Logger> logger;
        std::shared_ptr<Ice::Instrumentation::CommunicatorObserver> observer;
        std::function<void()> threadStart;
        std::function<void()> threadStop;
        std::function<void(std::function<void()>, const std::shared_ptr<Ice::Connection>&)> dispatcher;
        std::function<std::string(int)> compactIdResolver;
        std::function<void(const Ice::BatchRequest&, int, int)> batchRequestInterceptor;
        std::shared_ptr<Ice::ValueFactoryManager> valueFactoryManager;
    };
}
```

C++98

```
namespace Ice
{
    struct InitializationData
    {
        PropertiesPtr properties;
        LoggerPtr logger;
        Instrumentation::CommunicatorObserverPtr observer;
        ThreadNotificationPtr threadHook;
        DispatcherPtr dispatcher;
        CompactIdResolverPtr compactIdResolver;
        BatchRequestInterceptorPtr batchRequestInterceptor;
        ValueFactoryManagerPtr valueFactoryManager;
    };
}
```

C#

```
namespace Ice
{
    public class InitializationData : ICloneable
    {
        public object Clone() { ... }

        public Properties properties;
        public Logger logger;
        public Instrumentation.CommunicatorObserver observer;
        public System.Action threadStart;
        public System.Action threadStop;
        public System.Action<System.Action, Connection> dispatcher;
        public System.Func<int, string> compactIdResolver;
        public System.Action<BatchRequest, int, int> batchRequestInterceptor;
        public ValueFactoryManager valueFactoryManager;
    }
}
```

Java

```
package com.zeroc.Ice;

public final class InitializationData implements Cloneable
{
    @Override
    public InitializationData clone() { ... }

    public Properties properties;
    public Logger logger;
    public com.zeroc.Ice.Instrumentation.CommunicatorObserver observer;
    public Runnable threadStart;
    public Runnable threadStop;
    public ClassLoader classLoader;
    public java.util.function.BiConsumer<Runnable, Connection> dispatcher;
    public java.util.function.IntFunction<String> compactIdResolver;
    public BatchRequestInterceptor batchRequestInterceptor;
    public ValueFactoryManager valueFactoryManager;
}
```

Java Compat

```
package Ice;

public final class InitializationData implements Cloneable
{
    @Override
    public InitializationData clone() { ... }

    public Properties properties;
    public Logger logger;
    public Ice.Instrumentation.CommunicatorObserver observer;
    public ThreadNotification threadHook;
    public ClassLoader classLoader;
    public Dispatcher dispatcher;
    public CompactIdResolver compactIdResolver;
    public BatchRequestInterceptor batchRequestInterceptor;
    public ValueFactoryManager valueFactoryManager;
}
```

JavaScript

```
Ice.InitializationData = function()
{
    this.properties = null;
    this.logger = null;
    this.valueFactoryManager = null;
};
```

MATLAB

```
% in Ice package

classdef (Sealed) InitializationData
    properties
        properties_
    end
end
```

ObjC

```
@interface ICEInitializationData : NSObject
{
...
}
@property(retain, nonatomic) id<ICEProperties> properties;
@property(retain, nonatomic) id<ICELogger> logger;
@property(copy, nonatomic) void(^dispatcher)(id<ICEDispatcherCall>, id<ICEConnection>);
@property(copy, nonatomic) void(^batchRequestInterceptor)(id<ICEBatchRequest>, int, int);

-(id) init:(id<ICEProperties>)properties logger:(id<ICELogger>)logger
                                    dispatcher:(void(^)(id<ICEDispatcherCall>, id<ICEConnection>))d
                        batchRequestInterceptor:(void(^)(id<ICEBatchRequest>, int, int))i;
+(id) initializationData;
+(id) initializationData:(id<ICEProperties>)properties logger:(id<ICELogger>)logger
                                                    dispatcher:(void(^)(id<ICEDispatcherCall>,
id<ICEConnection>))d
                                     batchRequestInterceptor:(void(^)(id<ICEBatchRequest>, int, int))i;
// This class also overrides copyWithZone:, hash, isEqual:, and dealloc.
@end
```

PHP

```
namespace Ice
{
    class InitializationData
    {
        public function __construct($properties=null, $logger=null) { ... }

        public $properties;
        public $logger;
    }
}
```

Python

```
# in Ice module

class InitializationData(object):
    def __init__(self):
        self.properties = None
        self.logger = None
        self.threadStart = None
        self.threadStop = None
        self.dispatcher = None
        self.batchRequestInterceptor = None
```

Ruby

```
module Ice
    class InitializationData
        def initialize(properties=nil, logger=nil)
            ...
        end

        attr_accessor :properties, :logger
    end
end
```

For example, to set a custom logger of type `MyLogger` in C++, you could use:

C++11

```
Ice::InitializationData initData;
initData.logger = std::make_shared<MyLoggerI>();
auto communicator = Ice::initialize(argc, argv, initData);
```

or

```
Ice::InitializationData initData;
initData.logger = std::make_shared<MyLoggerI>();
Ice::CommunicatorHolder ich(argc, argv, initData);
```

C++98

```
Ice::InitializationData initData;
initData.logger = new MyLoggerI;
Ice::CommunicatorPtr communicator = Ice::initialize(argc, argv, initData);
```

or

```
Ice::InitializationData initData;
initData.logger = new MyLoggerI;
Ice::CommunicatorHolder ich(argc, argv, initData);
```

Back to Top ^

See Also

- Command-Line Parsing and Initialization
- The Properties Interface

Previous Next