Asynchronous APIs in JavaScript

Given JavaScript's lack of support for threads, Ice for JavaScript uses asynchronous semantics in every situation that has the potential to block, including both local and non-local invocations. Synchronous proxy invocations are not supported.

Here is an example of a simple proxy invocation:

JavaScript

```
const proxy = HelloPrx.uncheckedCast(...);
const promise = proxy.sayHello();
promise.then(
    () => {
        // handle success...
    },
    ex => {
        // handle failure...
    }
);
```

The API design is similar to that of other asynchronous Ice language mappings in that the return value of a proxy invocation is an instance of `Ice.AsyncResult`, through which the program configures callbacks to handle the eventual success or failure of the invocation. The JavaScript implementation of `Ice.AsyncResult` derives from the standard JavaScript Promise.

We can simplify the example above using the `await` keyword:

JavaScript

```
(async function()
{
    const proxy = HelloPrx.uncheckedCast(...);
    try
    {
        await proxy.sayHello();
        // handle success...
    }
    catch(ex)
    {
        // handle failure...
    }
}());
```

Note that the `await` keyword can only be used in a function marked with the `async` keyword.

Certain operations of local Ice run-time objects can also block, either because their implementations make remote invocations, or because their purpose is to block until some condition is satisfied. See Blocking API Calls for a list of these operations. Like proxy invocations, these operations return the standard JavaScript Promise. The example below shows how to call `ice_getConnection` on a proxy:

JavaScript

```
const communicator = ...;
const proxy = communicator.stringToProxy("...");
proxy.ice_getConnection().then(
    connection => {
        // got connection...
    },
    ex => {
        // connection failed...
    }
);
```