Doc No: N3327=12-0017
Date: 2012-01-12
Project: JTC1.22.32
Reply to: Niklas Gustafsson
Artur Laksberg
Microsoft Corp.
1 Microsoft Way
Redmond WA USA 98052-6399
Fax: +1-928-438-4456
Email: niklas.gustafsson@microsoft.com
artur.laksberg@microsoft.com

A Standard Programmatic Interface for Asynchronous Operations

While presenting a proposal that can be adopted or rejected in isolation, this document is related to N3328=12-0018. The reader is advised to read both as a unit and to consider how the two build on each other for synergy.

1. The Problem

With the increased ubiquity of application- and system architectures involving some form of network communication, generally resulting from the information technology industry's trend toward Internet-enablement, the importance of managing the inevitable latency of the communication operations becomes ever-more significant.

Communication-intensive code spends a large portion of its time waiting for I/O requests to complete or waiting for incoming requests to service. In such an environment, the efficiency of waiting is critical. To efficiently utilize the underlying hardware resources, which ultimately is about efficient energy and capital utilization, many programming environments offer libraries where I/O APIs either exclusively asynchronous or offer asynchronous variants of otherwise synchronous operations.

Indeed, there are non-standard libraries for C++ making such APIs available. One example is the Boost ASIO library of networking primitives. Another example is the C++ representation of the I/O libraries used with the next release of the Windows operating system, which rely on a completely different pattern than ASIO to represent asynchronous operations.

Such libraries thus exist in isolation and do not allow programmers to effectively compose applications comprising more than one library. Without a standard representation of asynchronous operations, programmers must choose to either build their own abstractions on top of existing libraries, or simply forgo combining libraries. Either decision represents a failure of the language to provide necessary facilities for modularity and composition.

With this limitation related to programming asynchronous operations, C++ suffers a deficit in programmer productivity compared to other languages, such as JavaScript, C#, Java, F#, Python, making it less productive for writing I/O-intensive application and system software than other languages. In particular, writing highly scalable services becomes significantly more difficult in C++ than in, for example, C# or JavaScript.

Adopting a standard representation of asynchronous operations will open up the doors for defining standard libraries for composing long-running computational work with various forms of I/O (networking, file, database, video, audio, serial devices, etc.) into highly scalable, responsive software components for use in both systems and application programming.

The functionality of std::future / std::promise offer a partial solution to the problem: it allows separation of the initiation of an operation and the act of waiting for its result, but the waiting act on the future is synchronous and there is no standard model for parallel or even sequential composition of futures.

This proposal outlines the necessary components of a rich representation of asynchronous operations, matching (and in many cases surpassing) the capabilities of other programming language environments. It is entirely based on library extensions and does not require any language features beyond what is available in C++ 11.

The proposal is an evolution of the functionality of std::future / std::promise, but is focused on the conceptual end result and does not get too caught up in the details of the evolutionary aspects or the details of such things as allocators and precise copy semantics. I return to the topic of evolution in section 3.1.

2. The Proposal

2.1 Terminology

When programming high-latency I/O operations, asynchronous APIs surface in two common and distinct patterns:

i. Asynchronous operations, where an action (computation, I/O operation, etc.) is explicitly initiated by application code and its progress may be observed by the application software. Typically, operations either produce a value, result in some side-effect, or both.

   Associated with an operation, or a set of operations, is a continuation, the definition of work to be started once the operation(s) is (are) completed.

   Operations may be canceled, which means that the initiator requests that the work of the operation be ceased at the first convenient point in time. An operation that has already been completed cannot, in most systems, be canceled.

ii. Asynchronous events, where a series of calls into the application code are generated as the side-effect of an initial API call made by the application software, registering with an event source an event handler, i.e. a callback function (or lambda) that is invoked once for each arriving event.

For example, an HTTP-based service may listen for new HTTP requests, which arrive as events, independent and unrelated of each other. The application responds to these events by sending information back to the requestor, i.e. the HTTP client.

[Note: This proposal concerns a standard model for representation of operations, not events. Events are usually adequately represented using straight-forward callback-based approaches and are mentioned here only to point out the important distinction between events and operations. Attempts to conflate the two, as some libraries do, typically do not result in models that allow for straight-forward composition of operations.]

Scheduling is the act of deciding what machine resources the work associated with an operation or event should be allocated. Scheduling may happen in the operating system, in supporting libraries, or (generally) both.

As an example of library-assisted scheduling, consider the limitation that many graphical user interface environments impose on application code: that any work manipulating graphical user interface elements must be coordinated (scheduled) by a single operating system thread.

2.2 Operations

All asynchronous operations are represented by a single template type ‘future<T>’, defined in the ‘std’ namespace. A template specialization ‘future<void>’ is used to represent operations that only have side-effects and no resulting value.

To support composition of libraries, all asynchronous operations should be represented as a future<T> (including future<void>): it is the intent of this proposal to strongly discourage derived types from being used in public APIs.

```
using namespace async;
using namespace std;

shared_ptr<char> buf;

future<int> rd = stream.read(512, buf);
```

Once a future has been created, its status may be queried by calling either of the member functions ‘is_done()’ or ‘was_canceled()’. The former may be used to poll for completion, while the latter is used to check whether a future has been successfully canceled. Cancellation will be discussed in further detail in section 2.8.

A future result value may be retrieved without risk of blocking using ‘try_get()’:

```
int bytes_read;
```

```
if ( rd.try_get(bytes_read) ) { ... }
```

Typically, however, a continuation will be associated with the future using the ‘then’ member function, taking a function object to which is passed the future. From within the function object, the future’s ‘get()’ function may be invoked to retrieve the value without risk of blocking:

```
auto pls11 = rd.then([](future<int> op) { return op.get() + 11; });
```

In this example, a new future is defined which will have as its value the value of the read operation plus eleven. The type of ‘pls11’ is future<int>.

2.3 Creation

Futures are constructed using factory functions declared in the ‘std’ namespace and on classes std::promise and std::packaged_task. There is also a (new) constructor on std::future<T> taking a value of type T.

The future factories correspond to three kinds of operations:

i. Values, i.e. operations for which the value is known at the point of construction. These are created using the functions std::create_value<T>() and std::create_void(). A value-future may also be constructed directly using std::future(const T&).

ii. Computational tasks, i.e. operations where the resulting value is computed by a long-running function used to define the operation. These are created using a std::packaged_task() and then calling ‘get_future()’ on the packaged_task instance.

iii. Futures, i.e. operations where the computation of a result is defined through some other means, such as an I/O request. These are created by first creating a std::promise and then calling ‘get_future()’ on the promise instance.

A function defined to return a future<int> may utilize all three of these constructors:

```
std::future<int> compute(int x)
{
    if ( x < 0 ) return std::create_value<int>(-1);
    if ( x == 0 ) return std::create_value<int>(0);

    if ( x > 1000000 )
    {
        promise<int> p;
        auto result = p.get_future();
        compute_on_cluster(p,x);
        return result;
    }
```

```
            return std::packaged_task<int()>(
                [x]()
                {
                    compute_locally(x);
                }).get_future();
        }
```

Futures are also created through composition, discussed in section 2.7.

2.4 Completion

Completion of a future is handled without direct programmatic involvement for tasks and values: tasks complete by returning from execution of the function object, while values are already complete when the constructor returns.

For futures, the completion is effected by calling either of these functions on the promise instance used to create the future:

```
    bool set_value(const T& val); // for future<T>
    bool set();                   // for future<void>
```

The return value indicates whether setting the future value succeeded or not.

There are two reasons why completion may not be successful:

i. The future is already complete.
ii. The future was canceled.

2.5 Value Retrieval

The value of a future is retrieved using get(), which is a potentially blocking function, or try_get(), which returns false if the value is not yet available:

```
    const T& get();
    bool try_get(T& ref);
```

The “doneness” of the future may also be tested using is_done(), which returns true if get() will not block (the future completed or was cancelled). For future<void>, where no value is produced, there is no need for a try_get().

2.6 Promptness

The presence of a future<T> as either an argument to or result of a function represents the possibility that the computation of the value may require blocking. However, there are many situations where an operation only requires blocking under some circumstances, or where the operation is fast enough that the value is computed before it is needed by the initiating code.

For example, a library may be buffering I/O requests so that only an initial request take a significant amount of time, while subsequent requests find the data already available. In another situation, it may be that a function produces a constant value in 90% of actual invocations, but have to compute its result using a long-running computation in 10% of its calls. To avoid blocking 10% of the time, a properly designed library will always return a future<int>, but fill it with the constant value before even returning the future object to the caller.

In such situations, the operation is said to finish promptly, and it occurs with enough likelihood that it is important to take into consideration in the design and implementation of std::future<T>.

The ability to test for early completion allows us to avoid associating a continuation, which needs to be scheduled with some non-trivial overhead and with a near-certain loss of cache efficiency.

The situation is addressed using try_get(), which will return a value if there is one already available, and is_done(), which communicates that a value is available without retrieving it.

These allow application code to avoid attaching the continuation:

```
future<int> rd = stream.read(512, buf);
int bytes_read;

if ( ! rd.try_get(bytes_read) )
{
  rd. then([](future<int> op)
  {
      int bytes_read = op.get();
      // do something with bytes_read
  });
  return;
}
// do something with bytes_read
```

try_get() and is_done() are generally subject to performance-related race conditions. Prompt operations do not suffer any race conditions; this allows for optimized try_get() and is_done() implementations that avoid taking locks when operations are prompt.

2.7 Composition

Futures may be composed with function objects to create new futures. The fundamental composition of a single future with a continuation function has already been introduced, but there are other forms of composition.

In all cases, the result of composing two or more futures and/or function object(s) is another future.

The following fundamental forms of composition are necessary:

i. Sequential composition, i.e. future a followed by b, where b is defined by a function object. The value of the combined future is the value of b. This has already been introduced in the form of the .then member function.

ii. Parallel AND composition, i.e. futures a and b are started and completed without explicit coordination, but joined into a single future that reflects the completion of both a and b: the combined future holds the value of a and b as a tuple or vector.

iii. Parallel OR composition, i.e. futures a and b are started and completed without explicit coordination, but joined into a single future that reflects the completion of either a or b: the combined future holds the value of the input future that completes first.

Additional composition operations can be built from these primitives, for example various forms of loops and aggregation operations.

2.7.1 Sequential Composition

Sequential composition takes future a and function object b, organizing them such that b is run after a completes and eventually returns the value of b as the value of the composed future.

In sequential composition, an antecedent future may be combined with a function returning a value of type T (T not being future<S>), in which case the result of the composition is future<T>; it may alternatively be combined with a function object returning a value of type future<S> (as above), in which case the resulting type is future<S>.

In this example, a read future from one source is followed by a write to a target, of the same number of bytes that were read, ultimately returning the number of bytes that were transferred:

```
auto buf = make_shared<std::vector<unsigned char>>();
auto streamR = make_shared<async_istream>(name1);
auto streamW = make_shared<async_ostream>(name2);

future<int> rd = streamR.read(512, buf);

auto wr = rd.then(
  [buf](future<int> op) -> future<int>
  {
      if ( op.get() > 0 )
      {
          return streamW.write(op.get(), buf);
      }
      return std::create_value<int>(0);
  });
```

2.7.2 Parallel AND

By virtue of their independent creation, independently created futures are naturally forked vis-à-vis each other, so parallel composition need only concern the joining of the results.

Under AND-composition, the resulting future holds the values of all input futures, presented as either a tuple (if the input futures are fixed in number and of heterogeneous types) or a vector (if the input cardinality is unknown at compile time and the futures are all of the same type).

Thus, we can easily start multiple read futures and continue work only once we have the results from all:

```
auto buf1 = make_shared<std::vector<unsigned char>>();
auto buf2 = make_shared<std::vector<unsigned char>>();
auto streamR1 = make_shared<async_istream>(name1);
auto streamR2 = make_shared<async_istream>(name2);
auto streamW = make_shared<async_ostream>(name3);

future<int> rd1 = streamR1.read(512, buf1);
future<int> rd2 = streamR2.read(512, buf2);

future<tuple<int,int>> wa = std::when_all(rd1, rd2);

auto wr = wa.then(
  [buf1,buf2]
  (future<tuple<int,int>> op) -> future<int>
  {
      tuple<int,int> res = op.get();

      if ( get<0>(res) > 0 )
      {
          return streamW.write(get<0>(res), buf1);
      }
      if ( get<1>(res) > 0 )
      {
          return streamW.write(get<1>(res), buf2);
      }
      return std::create_value<int>(0);
  });
```

2.7.3 Parallel OR

Under OR-composition, the resulting future represents the value of the first if the input futures to complete. All input futures must be of the same type, future<T>.

This is a non-deterministic choice operator.

Thus, we can start multiple read futures and continue work once we have a result from one of the futures. The index of the future that finished first is held in the first element of a tuple<int,T>, while the second element holds the result of the finished future.

```
auto buf1 = make_shared<std::vector<unsigned char>>();
auto buf2 = make_shared<std::vector<unsigned char>>();
auto streamR1 = make_shared<async_istream>(name1);
auto streamR2 = make_shared<async_istream>(name2);
auto streamW = make_shared<async_ostream>(name3);

future<tuple<int,int>> wa = std::when_any(rd1, rd2);

auto wr = wa.then(
  [buf1,buf2]
  (future<int> op) -> future<int>
  {
      int idx = get<0>(op.get());

      if ( get<1>(op.get()) > 0 )
      {
          return streamW.write(get<0>(res), (idx == 0)?buf1:buf2);
      }
      return std::create_value<int>(0);
  });
```

2.8 Cancellation

By having the future object represent not just the result of the operation, but the operation itself, additional properties and features can be meaningfully associated with the interface. One such is cancellation, which is used to prevent an operation that has not yet started from starting, or cause the early cessation of an operation that is already under way.

```
future<int> rd = stream.read(512, buf1);

rd.cancel();
```

A canceled future cannot provide a result value, either via try_get(), get() or through composition. Attempts to retrieve its value using the functions will result in an std::canceled_task exception being thrown.

An already completed future cannot be canceled.

Fundamentally, cancellation is subject to race conditions: a future that returns false from a call to is_done() may be done by the time a subsequent call to cancel() is issued. Cancelling a future may also result in its side-effects either having been performed or not.

Cancellation is supported by the use of cancellation tokens, objects that act as publishers in a simple pub/sub pattern: the token acts as a pub/sub topic, to which listeners (typically futures) may register as subscribers.

Each future has its own private cancellation token, to which only the future itself is a subscriber. Futures may further be associated with an externally created token and listen to its cancellation requests.

Invoking the cancel() function on a future cancels only that one future and any of its continuations, whereas cancelling an externally created token will cancel all associated futures that haven't already completed. This is the case even for futures that have been associated with an external token.

This may be combined with parallel OR composition to stop additional work once the first future is finished:

```
cancel_token token;

future<int> rd1 = streamR1.read(512, buf1, token);
future<int> rd2 = streamR1.read(512, buf2, token);
future<int> rd3 = streamR1.read(512, buf2, token);

future<tuple<int,int>> wa = std::when_any(rd1, rd2, rd3);

auto wr = wa.then(
  [buf1,buf2,token]
  (future<int> op) -> future<int>
  {
      token.cancel();

      int idx = get<0>(op.get());

      if ( get<1>(op.get()) > 0 )
      {
          return streamW.write(get<0>(res), (idx == 0)?buf1:buf2);
      }
      return std::create_value<int>(0);
  });
```

The cancel() function returns true if the token (or future) is found in the canceled state upon function return (i.e. it was successfully canceled or already canceled).

2.9 Propagation of Exceptions

Exceptions raised during the evaluation of an operation are propagated into the future object and re-thrown when try_get() or get() are called.

In the case of a task created using a promise, the promise method set_exception() may be used to explicitly communicate which exception to propagate. In the case of a task defined using packaged_task<T>, any exception propagated from the invocation of its function object will be used as the propagated exception. In the case of a task created using create_value<T>(), the exception object has to be passed into the method when creating the value.

A call to set_exception() is regarded as a completion of the future, just as if set_value() had been called: is_done() will return true and cancellation will fail.

When combined with other futures, futures with exceptional values will generally not be treated specially: the future is passed from the antecedent to the continuation and an exception raised when get() or wait() is called.

Special handling is required for:

i. when_all(), where the exceptions held by any input future is transferred to the resulting future. If more than one input future completes with an exception, one of them is picked, non-deterministically.

ii. when_any(), where the exception raised by the winning input future is transferred to the resulting future.

2.10 Scheduling

In most cases, explicit scheduling of asynchronous operations and their continuations is not problematic: continuations may be executed by the thread that completes the future, or scheduled on a global thread pool.

There are circumstances, however, where more precise control over resources, whether hardware or software, is necessary. Unfortunately, std::thread class does not provide the abstractions necessary for such control.

In such circumstances, futures and packaged_tasks may be associated with an instance of the abstract type std::scheduler, which has the following definition:

```
class scheduler_base
{
public:
    virtual void run() = 0;
};

class scheduler
{
public:
    virtual void post(std::task_base oper) = 0;
    };
```

The class task_base is a public abstract type available for implementations of future<T> to use for scheduling purposes.

The circumstances when explicit scheduling may be necessary include, but are not limited to:

i. std::packaged_task<T> needing a scheduler for its computation.
ii. The continuation of a future requires significant time to complete and therefore cannot necessarily execute in the context of the completing thread (inline). Association of a scheduler with the future causes the future's continuation to be scheduled using the scheduler.
iii. A continuation needs to execute on the thread owning a particular hardware or software resource, such as the main graphical user interface thread.
iv. The application needs to throttle its work into a thread pool of a limited size.

Association of a scheduler with a future is done at its construction time using one of the overloaded functions / constructors that take a scheduler reference.

3. Interactions and Implementability

3.1 Interactions

The definition of a standard representation of asynchronous operations described in this document will have very limited impact on existing libraries, largely due to the fact that it is being proposed exactly to enable the development of a new class of libraries and APIs with a common model for functional composition.

Few libraries within the existing ISO standard for C++ are designed for asynchronous usage. Non-standard libraries may need to be redesigned in order to participate in the compositional model offered by the standard representation, or continue to exist in isolation of other libraries.

As asserted in the introduction, adopting a standard representation of rich asynchronous operations should open up the doors for defining standard libraries for composing long-running computational work with various forms of I/O (networking, file, database, video, audio, serial devices, etc.) into highly scalable, responsive software components for use in both systems and application programming.

In moving forward with this proposal, it will be necessary to more deeply examine the details of how the C++ 11 definitions of std::future, std::promise and std::packaged_task can be evolved into the form outlined in this proposal.

3.2 Implementability

This proposal is based on, but not identical to, a library developed at Microsoft which will ship with the next release of Visual Studio. The delta between what is proposed and what is implemented and working mostly relates to implementation details and opportunities to optimize in the presence of promptness. The names are also different, as our implementation of std::future is based on the standard.

We thus have evidence that the features described can be implemented and be high-performing.

The library in question is being adopted by several groups across Microsoft and will be a key component of how the next release of Windows presents I/O-based operations to application developers using C++. In fact, our implementations of std::future, std::thread, etc. is based on the library discussed.

3.3 Implementation Issues

When moving from a synchronous formulation of an algorithm to an asynchronous one, a major issue is the impact on object lifetime management that the move has, and it impacts the implementation, too.

In order to free up expensive resources, primarily threads, while waiting for operations to complete, the call stack typically has to be unwound all the way to where no user frames are active so that other code can be scheduled on the thread. Exceptions to this rule include systems that don't rely on traditional contiguous program stacks for their activation frames: for the vast majority of mainstream runtime environments, the rule that the stack must be unwound applies.

Typical code will therefore look something like this:

```
void copy(std::ios::stream strR,
          std::ios::stream strW,
          size_t size)
{
   shared_ptr<char> buf(new char[size]);

   future<int> rd = strR.read(size, buf);

   auto wr = rd.then([buf,strW](future<int> op) -> future<int>
     {
         if ( op.get() > 0 )
         {
             return strW.write(op.get(), buf);
         }
         return std::create_value<int>(0);
     });

   return;
}
```

After setting up the operation and its continuation, the function returns. Any local objects will typically be destroyed well before the continuation is invoked. If the stream objects passed in are allocated as local variables by the caller, they, too, will be destroyed before the continuation is invoked, because the caller has to be unwound, too.

It is therefore of critical concern that objects that are to be used with asynchronous operations are designed to handle lifetime management in a more automatic fashion, e.g. by relying on a reference-counting scheme. This requires library designers of library classes to take this into account or users to make liberal use of shared_ptr<T> everywhere, preferably using calls to the efficient std::make_shared() function.