On the Future of std::future and a Future concept and Data-flow Programming

Hartmut Kaiser, Thomas Heller, Peter Sommerlad

2014-02-13

Document Number:	DXXXX
Date:	2014-02-13
Project:	Programming Language C++

1 History

1.1 Discussion on c++std-parallel mailing list

In 2013 there have been several discussions raised by papers **(find numbers)** that asked for extending `std::future` API with a member function `futre::then()` that allows to specify a function that will run after the future object becomes ready. The invocation of .then() would then return a future wrapping the original future object, etc.

Peter strongly objected to the abstraction of future gain "fat" by giving it more than the semantic of a *"ticket for a value or exception to be obtained later"*. While a concrete implementation such as std::future in the world of C++11 requires some hooking to a synchronization mechanism, the abstraction should be agnostic about where the value it eventually receives comes from. His colleague Luc Blser also supports his opinion and provides examples how chaining of tasks could be achieved with existing C++11 mechanisms.

To quote Luc's concerns:

> I also tend to share your standpoint, namely that futures are probably not the most appropriate concept for incorporating continuations. Futures and tasks are generally two different concepts: a future represents a value of possibly future computation, while a tasks abstracts a unit of asynchronous or parallel execution. Therefore, as you also stated, the future can but does not necessarily have to represent the result of a parallel or asynchronous task.
>
> Chaining continuations on futures may thus lead to the following consequences:

- Conceptual confusion: Futures would need to define an underlying task model, i.e. a synchronization point and thread pool mechanism. The notion of future would thus be subtly altered to the abstraction of a task, i.e. deviate from the common terminology.
- Implementation restriction: The two fairly separate concerns, namely result access and task chaining, are coupled in one interface, thus limiting degrees of freedom for implementation.

It would be probably more flexible if proper full-fledged tasks could be chained (each task having a future), or, if asynchronous operations are defined to be triggered on some available futures: i.e. something like async(future1, future2, [] (result1, result2) =¿ continuation). The latter is naturally already supported without syntactic extensions, i.e. async([]() =¿ x = future1.get(); y = future2.get(); continuation(x, y);). A benefit of this could be that the chaining of a single or multiple preceding tasks is uniformly supported. Multiplexed selection of ”any” first available futures could be possibly enabled by providing an any-combinator of futures. future3 = any(future1, future2).

Others seem to have the perspective that a `std::future` actually is about synchronization and thus chaining execution of code with respect to the event of a `std::future` instance becoming ready is the way to provide an attractive style of ”continuation-based” programming **(check terminology)**.

In addition std::future doesn't fit the needs for any task-based parallelism, because its tight coupling to std::promise and its heavy-weight.

2 Introduction

All the above issues led to the discovery by Olivier Giroux while discussing with Peter that there is a need for a concept *Future* for representing the ”later value” that is more light-weight than std::future and a separate concept for the synchronization aspect of std::future that comes close to a ”task completion” that might happen to also be supported by std::future or not.

2.1

2.2 Open Issues to be Discussed

2.3 Acknowledgements

Acknowledgements go to

3 Proposed Library Additions