Document No: WG21 N4551
Date: 2015-08-13
References: ISO/IEC PDTS 19571
Reply To: Barry Hedquist <beh@peren.com>
INCITS/PL22.16 IR

National Body Comments

ISO/IEC PDTS 19571

Technical Specification: C++ Extensions for Concurrency

Attached is WG21 N4551, National Body Comments for ISO/IEC PDTS 19217, Technical Specification – C++ Extensions for Concurrency.

Document numbers referenced in this document are from WG21 unless otherwise stated.

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
JP1	2	2.3	10	ge	The first sentence of the note seems incorrect. It says "the validity of the `future` returned from `then` cannot be established" but the `future` returned from `then` should be always valid according to the postconditions. Is "`then`" in the sentence actually "continuation(func)"?	Note: In case of implicit unwrapping, the validity of the `future` returned from ~~`then`~~ `func` cannot be established until after the completion of the continuation.	
JP2	2	2.4	10	ge	The same comment as JP1 for `shared_future`.	Note: In case of implicit unwrapping, the validity of the `future` returned from ~~`then`~~ `func` cannot be established until after the completion of the continuation.	
JP3		2.7	2	te	The return type of `when_all` is fixed to `std::vector` in the current proposal. Generalizing it to arbitrary sequence container by passing it as a template parameter may provide more flexibility for the users (e.g., use of a custom allocator.)	template <class InputIterator, class Container = vector<typename iterator_traits<InputIterator>::value_type> > future<Container> when_all(InputIterator first, InputIterator last);	
JP4	2	2.7	5	te	The description should be changed to match the change proposed by JP3.	A new shared state containing a Sequence is created, where Sequence is either ~~vector~~ sequence container or tuple based on the overload, as specified above.	
JP5	5	2.7	5	te	The description should be changed to match the change proposed by JP3.	If the first overload is called with first == last, when_all returns a future with an empty ~~vector~~ sequence container that is immediately ready.	
JP6		2.9	2	te	The same comment as JP3 for `when_any` (to parametarize the return type sequence.)	template <class InputIterator, class Container = vector<typename	

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
						iterator_traits<InputIterator>::value_type> > future<when_any_result<Container>> when_any(InputIterator first, InputIterator last);	
JP7	2	2.9	5	te	The description should be changed to match the change proposed by JP6.	A new shared state containing when_any_result<Sequence> is created, where Sequence is a ~~vector~~ sequence container for the first overload and a tuple for the second overload.	
JP8	2	2.9	5	ed	The expression in “a vector for the first overload and a tuple for the second overload” differs from “either vector or tuple based on the overload” in 2.7 paragraph 5. They should be uniformed because they say the same thing.	Not sure which is better in English. It’s up to the editor.	
JP9	6	2.9	5	te	The description should be changed to match the change proposed by JP6.	The futures field is an empty ~~vector~~ sequence container.	
GB 1	Page 15	3.4		Te	count_down(n) does not make sense when n > 1 It is stated that n ≥ counter, but a client does not know the value of the counter. Only is_ready() can tell whether count_down(n) is viable or not when n is > 1.	It would be reasonable to have a function, called get_counter(), that will return the current value of the counter. In addition, I suggest that count_down(n) should probably return min(counter,n). The return value is the actual value that is subtracted from the counter. Example: If the counter is 8 and one of the threads calls count_down(10), this call will return 8 and the value of the counter will become 0.	
GB 2	Page 17	3.6	P10	Te	The semantics of arrive_and_drop are unclear. The concurrency TS gives the effects of arrive_and_drop as:	Add a sentence to make it clear under what circumstances the choice is made, and what it means to remove the thread from the set without "arriving" at the barrier.	

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
					Either arrives at the barrier's synchronization point and then removes the current thread from the set of participating threads, or just removes the current thread from the set of participating threads. It is not clear how this choice is made, or what it means to "just remove the current thread from the set of participating threads". If that thread isn't considered to have "arrived", how are the waiting threads ever supposed to be released?		
GB 3	Page 18	3.9		Te	What is the meaning of "flex" in flex_barrier? It's unclear how the name of the class relates to its functionality.		

GB 4	Page 20	4.3		Te	atomic<T> has two overloads of each compare-exchange function for non-volatile values:
					
					bool compare_exchange_weak(T&,T,memory_order,memory_order);
					bool compare_exchange_weak(T&,T,memory_order);
					
					The concurrency TS makes that 4 for atomic_shared_ptr:
					
					bool compare_exchange_weak(shared_ptr<T>&,shared_ptr<T> const&,memory_order,memory_order);
					bool compare_exchange_weak(shared_ptr<T>&,shared_ptr<T>&&,memory_order,memory_order);
					bool compare_exchange_weak(shared_ptr<T>&,shared_ptr<T> const&,memory_order);
					bool compare_exchange_weak(shared_ptr<T>&,shared_ptr
						Either:
						
						Change the signatures back to match atomic<T>, taking the new value by value rather than by reference.
						
						Document the expected characteristics of the different overloads.	

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial

MB/ NC[1]	Line number	Clause/ Subclause	Paragraph/ Figure/Table	Type of comment[2]	Comments	Proposed change	Observations of the secretariat
					<T>&&,memory_order); However, there is no description of the difference in semantics. Presumably, the difference is that in the overloads with rvalue references the operation can move from the rvalue. However, without a semantic description it is not clear at what point it can do that: should it only move on success, or may the implementation always move, even on failure? Is it *required* to move on success, or may it always copy anyway?		
GB 5	Page 20	4.3		Te	atomic<T> has volatile overloads for every member function. atomic_shared_ptr and atomic_weak_ptr are missing those overloads.	Add volatile overloads for every member function to atomic_shared_ptr and atomic_weak_ptr	
GB 6	Page 21	4.3		Te	The concurrency TS lists the assignment operator from a shared_ptr as atomic_shared_ptr& operator=(shared_ptr<T>) noexcept; The atomic template in C++11 has T operator=(T) noexcept; This is so that the returned value can be used without having to reload from the atomic. (A similar signature is also used for atomic_weak_ptr.)	Change the assignment operator to shared_ptr<T> operator=(shared_ptr<T>) noexcept;	

1 **MB** = Member body / **NC** = National Committee (enter the ISO 3166 two-letter country code, e.g. CN for China; comments from the ISO/CS editing unit are identified by **)
2 **Type of comment:** **ge** = general **te** = technical **ed** = editorial