

ARC Counter Library Requirements and Design *

Markus Nordén

June 26, 2007

Abstract

This is a draft summary of requirements and a proposition of an API for the counter library of the ARC HED. Comments are encouraged and expected. In particular, feedback is needed on globality (section 2.4), persistency (section 2.5) and the suggested API (section 3).

Please, send any feedback to: `nordugrid-discuss@nordugrid.org`

1 Introduction

During the “Uppsala HED Implementation F2F Meeting”, it was decided that a counter library shall be provided for services and other components to use for e.g. allocation and housekeeping of resources.

This document contains requirements on that counter library based on e-mail discussions on the `nordugrid-discuss` e-mail list.

2 Requirements

2.1 Purpose

The purpose of a counter is to provide housekeeping of a resource that can be allocated and deallocated such as e.g. disk space or network bandwidth. The counter itself will not be aware of what kind of resource it limits the use of. Neither will it be aware of what unit is being used to measure that resource. Counters are thus very similar to semaphores.

The users of the resource must thus first call the counter in order to allocate (make a reservation of) an appropriate amount of the resource, then

*This document was used as input for discussions during NorduGrid Technical Meeting in Lübeck May 2–4, 2007. For the final design, see the Doxygen documentation of <http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/libs/counter/Counter.h>

use the resource and finally call the counter again to deallocate what was previously allocated.

Furthermore, the users of the resource must agree on what unit they use to measure the resource, e.g. if disk space is measured in GB, MB, kB or single bytes. The counter itself will thus only contain a number.

2.2 Counter types

Integers provide sufficient resolution for internal representation of available resources. Floating point numbers could be an alternative, but no such requirement has been expressed.

If there should arise a need to represent quantities of a resource that are smaller than one unit, the choice of unit for the corresponding counter can be reconsidered.

2.3 Atomicity and Synchronization

Counters must be able to handle concurrent operations in a consistent manner. This implies that all operations must be atomic. Furthermore, no operation may be performed until it is guaranteed that it will succeed, i.e. a call to the allocation method must block until there is a sufficiently large amount of the resource free.

No requirements for simultaneous operations on sets of counters have been requested.

2.4 Globality

Counters for different purposes (limiting different kinds of resources) may need to have different scopes:

Single thread The counter can only be accessed from within a single thread. There is no concurrency or need for synchronization and a special counter class is superfluous.

Single process (several threads) The counter can be accessed from all threads within a single process, but is not accessible from outside that process.

Single computer (several processes) The counter can be accessed from all processes running on CPUs with access to a common shared memory. This requires inter-process communication.

Single cluster (several computers) The counter can be accessed from any process running on any computer within a cluster.

Entire grid The counter can be accessed from any process running on any computer connected to a certain grid – probably by means of a counter service.

It deserves to be mentioned here that the scope of the counter itself is not necessarily the same as the scope of the resource it limits the use of. For example, there may be counter available only on the front end machine of a cluster that keeps track of the number of free CPUs in the entire cluster. This counter can be used by a process that runs on the front end machine but starts jobs on other computers in the cluster.

Requests have been made for single process counters, single computer counters and single cluster counters.

2.5 Persistency

Different kinds of counters have different lifetimes. The lifetime of single computer counters is an open question.

Process lifetime The counter is initialized at process start up and vanishes when the process finishes. Applicable only for single process counters.

System lifetime The counter is initialized at system start up and is reset upon system restart. Possibly applicable for single computer counters.

Eternal lifetime The counter is initialized once and is then available until it is explicitly removed. Applicable for single cluster, entire grid and possibly also for single computer counters.

2.6 Identification of counters

Counters are identified by unique names. The form of a name will depend of the globality of a counter.

Single process counters may, for example, be implemented as global objects accessible from all threads of the process. The “name” of a counter will in this case be the address of the corresponding counter object. This address can be referred to by an arbitrary number of pointers and references as well as one or zero “ordinary variables”, i.e. just as any other object.

A single computer counter or a single cluster counter may be accessed through an arbitrary number of counter objects in different processes. There may thus be many counter objects that correspond to the same counter and manipulates it in a synchronized way. Upon construction of such a counter object, the name of the counter shall be provided. Such names can follow the same rules as e.g. identifiers in the C programming language. Furthermore, the name of some global object, e.g. a file, will probably also be required as an “environment” for the counters.

The name of an entire grid counter will probably consist of the URI of a counter service (the “environment” part) and a counter name as described above.

2.7 Priority

In some cases it may be desirable to let important tasks exceed the limit of a counter to a certain extent. This may be implemented by means of a credit. That an allocation is prioritized may be expressed by means of an extra, optional, parameter to the allocation method.

2.8 Time Awareness

The basic use of a counter is that a certain amount of it is acquired by a call to an allocation method and subsequently returned by a call to a deallocation method.

There has also been requests for self expiring allocations, i.e. that it is specified at allocation for how long that acquisition shall last. After that time has elapsed, deallocation will occur automatically.

It has also been mentioned that reservation of resources in advance would be a nice feature, but this is considered to be less important.

3 Design and API

There will be an abstract base class that defines the interface of a counter, i.e. what operations can be performed on it. The different kinds of counters (different globality) will be implemented as subclasses of that base class.

The signature of the constructors of the different classes will be determined later based on the needs of the implementation.

The following operations will be available for counters:

3.1 void wait(int amount, bool prioritized, int duration)

Blocks until the requested amount is free and allocates it.

Parameters:

amount The amount to be allocated.

prioritized True if this is a prioritized request, false otherwise. Optional, default value is false.

duration Zero for allocations that subsequently will be explicitly deallocated. If positive, duration of a self expiring allocation. Optional, default value zero.

3.2 `bool tryWait(int amount, bool prioritized, int duration)`

Tries to allocate the requested amount and returns immediately. Returns true if the allocation succeeded and false if it did not succeed.

Parameters:

amount The amount to be allocated.

prioritized True if this is a prioritized request, false otherwise. Optional, default value is false.

duration Zero for allocations that subsequently will be explicitly deallocated. If positive, duration of a self expiring allocation. Optional, default value zero.

3.3 `void post(bool amount)`

Deallocates a certain amount.

Parameters:

amount The amount to be deallocated.

3.4 `int getLimit()`

Returns the current limit of the counter.

3.5 `int setLimit(int limit)`

Sets the limit of the counter. Returns the new limit.

Parameters:

limit The new limit.

3.6 `int changeLimit(void amount)`

Changes the limit of the counter by a certain amount. Returns the new limit.

Parameters:

amount The amount to add to the the limit. May be positive or negative.

3.7 `int getCredit()`

Returns the current credit of the counter.

3.8 int setCredit(void credit)

Sets the limit of the counter. Returns the new limit.

Parameters:

credit The new credit.

3.9 int changeCredit(void amount)

Changes the credit of the counter by a certain amount. Returns the new credit.

Parameters:

amount The amount to add to the the credit. May be positive or negative.

3.10 int getValue()

Returns the current value of the counter, i.e. the number of units currently available for allocation.

4 Implementation

The implementation will be based on “lower level” libraries. Posix and System V semaphores have been considered so far but have been rejected due to limited functionality and portability issues. The current candidate is glibmm, possibly combined with file storage for counters that cannot reside in memory.