Problem

Design and implement a generic mechanism to limit the number of messages sent within a one-second timeframe. A solution which can be reused for unrelated but similar purposes is preferred.

Background

It is common practice for exchanges to limit the number of messages each participant may send on a transactions-per-second basis. The TPS limit includes order messages, amendments, and cancellations. As an example, a market may institute a rule that your trading system will be allowed to send 100 transactions-per-second. Any messages in excess of the limit will be rejected. If too many rejections are generated the session may be terminated or the exchange might even levy a substantial fine. The exact figure of 100/sec may change in the future.
Currently messages tend to be sent in short but intense bursts. A solution is needed to slightly delay messages in excess of the limit to prevent violating market rules without rejecting messages.
It is also important to treat some messages as more urgent than others. For instance, cancel messages should be carried out before new order requests.
You may want to use this same implementation elsewhere in your trading system (such as a message throttle between two busy components), so a generic reusable design would be a great benefit.

Requirements

1. The solution shall use a sliding window calculation for transactions-per-second. This means that at any given point in time, the amount of transactions sent in the last 1000 milliseconds must be considered. This is opposed to a fixed window method where the count is reset on seconds boundaries based on the wall clock.

2. Messages in excess of the given limit (100/sec) must be queued
a. The solution must delay only for the minimal amount of time required to be inside the limit. For example, if 2 messages were sent every 10 milliseconds for the last 500 milliseconds, the next order should be delayed no longer than 10ms (you can assume the machine has a very precise clock).

b. Messages inside the limit should be sent immediately with minimum delay

3. Any structure can be used for messages – this is left to the imagination of the implementer and is not considered relevant to the problem. Use whatever makes the solution clearer or cleaner

4. When queued, cancel messages must be sent before other messages similarly queued

a. Messages should otherwise be sent in the same order they were queued

5. When some arbitrarily large number of messages have already been queued, the solution may start outright rejecting additional messages

6. Optimize for the best case scenario – the majority of orders are not expected to queue

Your sample message could be:

```
struct  NewOrder
{
  char id_[12];
  char side_; //'B' for Buy 'S' for Sell
  int price_;
  int size_;
  char symbol_[10];
};


struct  AmendOrder
{
  char id_[12];
  int price_;
  int size_;
};


struct  PullOrder
{
  char id_[12];
};
```