# Order timeout in SC-2

*Updated: 2023-05-09, for SC-2 ver. 1.008*

This documents discusses **order timeout**, an essential element of the SC-2 application. It is used to ensure proper handling of the situations when a pool issues a replenishment order, but the order is never filled, or not filled within a reasonable timeframe.

## Why we need to keep track of outstanding orders

In SC2, as in SC1, many pools (including the input buffers of various production units provided with a "safety stock" capability) operate on the Make-to-Stock (MTS) principle, which can be described as follows: the pool has two constants, the reorder point and the target level. During a regular inventory check (say, every morning), if the inventory level is found to be below the reorder point, the pool sends out a replenishment order to its supplier, the order amount being equal to

targetLevel - currentInventory

One key point, however, is missing from the above description. Once a replenishment order has been sent out, it will take certain amount of time (at least a few days) for the ordered product to arrive. Therefore, if the above reordering rule is literally exercised, for example, every day once a day, then the pool will end up making a new order every day. If it takes, say, 10 days for the first replenishment shipment to arrive, then we will end up making 10 orders, this ordering 10 times as much product as we really wanted!

Clearly, the MTS scheme has to take into account the orders that have been made but [whose shipments] have not arrived yet. The rest of this documents discusses how these **outstanding orders** are managed in SC2.

## Two ways to keep track of outstanding orders

There are at least 2 obvious ways to manage outstanding orders.

**(A) "One order at a time" (OOAAT).** A simple approach, mentioned by Ben in one of the recent Zoom calls, is "One order at a time". That is, once a pool has sent out a replenishment order, it should not make any more orders until the first order has been filled (i.e. the shipment has arrived).

**(B) "Managing the OnOrder Amount" (MOOA).** A slightly different approach has been used in SC1 and SC2, which we can call "Managing the OnOrder Amount" (MOOA). It works as follows:

- Throughout the simulation, the pool receiving order maintains an amount called `onOrder`, representing the orders that have been transmitted to the supplier, but whose shipments have not arrived yet. The `onOrder` amount is updated as appropriate whenever an order is issued, and whenever a shipment arrives.
- During the daily inventory check the `onOrder` amount is added to the current stock, as if the ordered shipment has already arrived. Thus the criterion for sending an order is that

  currentLevel + onOrder ≤ reorderPoint,

  and the ordered amount is

  targetLevel - (currentLevel + onOrder).

As we will see later, this MOOA approach can be further finessed by maintaining a list of all placed and not yet filled, and computing the `onOrder` amount as the sum of the sizes of these orders.

**Comparing the two approaches.** In general, OOAAT can be viewed as a stricter verson of MOOA, with no new ordering being done until `onOrder==0`.

The two approaches, OOAAT and MOOA, are equivalent in certain simple special cases. In particular, it is easy to see that they are equivalent when `reorderPoint < 0.5 * targetLevel`, and every order arrives in a single shipment.

The two approaches, however, behave differently in other circumstances. For example, if `reorderPoint ≥ 0.5 * targetLevel`, and the shipment time is long (longer than the time during which the amount of the shipment is typically consumed), the MOOA approach would allow the pool, very sensibly, to sometimes make a second order before the first one has arrived. (As an example, imagine a restaurant that buys its favorite wine by mail, ordering one 12-bottle case at a time. [E.g. their targetLevel=30 and reorderPoint=24]. It takes about 10 days for an order to arrive; but the restaurant typically sells about 20 bottles in a 10-day interval. Thus it makes sense for them to often have 2 outstanding orders simultaneously.) The MOOA approach is also more reliable if an order is sometimes fulfilled in multiple shipments, with the last shipments sometimes being very slow; unlike OOAAT, MOOA won't wait for the last shipment when it becomes counterproductive.

## Why order timeout is needed

The procedure for keeping track of outstanding orders, as described above (either OOAAT or MOOA) preventsthe MTS policy from placing unnecessary duplicate orders. However, it also makes the MTS ordering system vulnerable to be permanently disabled by certain types of disruptions, even though the disruptions themselves are transient.

The disruptions in question are those that don't merely delay a shipment, but result in an ordered shipment never arriving (or arriving only partially). This includes:

- Shipment Loss. (Order transmitted; shipment sent; it never arrives)
- Information Loss. (The customer thinks it has a valid order sent, but the supplier never receives it and never ships anything).
- Depending on the supply chain design, an Adulteration or Stock Depletion occurring somewhere betweenthe point to which the order is sent and the destination pool may have this effect as well. As an example, consider a system in which a finished-product Pool A sends an order to a raw material supplier, and the production nodes forming the chain from the RM suplier to Pool A are input-driven. (That is, they don't receive explicit orders themselves, but simply process all materials given to them). Normally, the orders are sized so that Pool A knows how many units of finished product it will receive for a given order sent to the RM supplier. But if more material than usual is used in the interediate stages (e.g. due to greate discard rate at QA stages), then the expected amount of finished product never arrives to Pool A.

When an order never arrives, the `onOrder` amount is never decremented; this is equivalent to reducing the `targetLevel` by the size of the order (or the size of the missing part of the shipment). Eventually, if enough orders fail to arrive, the `onOrder` amount becomes as large as `targetLevel`, which means that the MTS process stops placing new orders. Thus a few lost shipments (whose size totals to `targetLevel`) will result in the MTS system for the Pool being permanently broken! This may entirely break the supply chain, depending on how the pool is connected in it.

A simple cure for this problem, which is used in SC2, is the **order timeout**. It is implemented as follows: the pool that sends orders (Pool A) keeps track of each orders it has placed, storing the timestamp and the outstanding (not-yet-filled) amount of each one. Whenever a shipment arrives, it is applied to the the oldest order, reducing the outstanding amount of this order, or entirely removing the order from the list once it has been completely filled. The `onOrder` amount is computed dynamically as the sum of the current sizes of all currently outstanding orders.

Every day the system checks the oldest outstanding (unfilled) order in this list; if it is older than a certain number of days (the timeout interval for this pool), then we say that the order has **timed out**, and remove it from the list. This results in the reduction of the current `onOrder` amount, and enables the MTS system to place another order, if needed. In this way, disruptions such as shipment loss or order messsage loss only disable the MTS system for the duration of the timeoout interval, rather than forever.

In the current SC2 model implementaion, the timeout intervals for each MTS pool is specified by the parameter `orderExpiration` in the configuration file (`sc2.csv`). When a pool orders from multiple sources, it would probably make more sense to specify timeout intervals individually for each supply route, but for simplicity we currently (ver 1.008) associate the same timeout interval with all supply routes of a pool.

**Choosing the timeout interval.** The timeout interval for each pools has been chosen manually based on our expectation of the "normal" filling times.

For example, the DP orders from the DC, and it is expected that the DP is normally able to fill each order in a single shipment. Thus the timeout interval for this order is set to be just a bit longer than the upper bound of the distribution from which the shipping time for this order is taken.

On the other hand, the HEP places some orders with the DP, and the size of these orders exceeds the DP's normal stock level (its target level), which means that the DP itself will need to replenish itself twice before it can fully fill the HEP's order. We therefore set the timeout interval for the HEP's order to be sufficiently long to accommodate this choreography of back-and-forth exchange of messages and shipments.

The way the DC places orders is quite different from the way the downstream pools do. The DC's order does not go to another pool that is expected to be able to immediately initiate a shipping. Instead, the DC's order, mediated by MedTech, merely initiates the production order in a long supply chain, starting from an order for a raw material. Therefore, the timeout interval for the DC's order is set based on our expectation on how much time it would take for a typical production order to be filled, with the materials working their way through all the production stages.

## Why we want to cancel timed-out orders

By enabling an MTS pool to "time out" unfilled orders, we have protected it from becoming unable to order any more product because of an order loss. However, we have created an opposite problem: the pool now can order *too much*. That is, if no product arrives to the pool for an extended period of time, the pool will time out an old order, and send its copy to the supplier again. Now, this is not a problem if the orders weren't filled because of shipment loss or message loss. But what if the orders weren't filled on time because the supplier for some reason wasn't able to send anything (e.g. having an empty warehouse due to a production halt disruption somewhere upstream), but kept the receiving orders on its order book? Then, when the product becomes available, it will try to fill all these back orders, potentially flooding the recipient with much more product than it needs.

This is particularly pernicious if Node A (a production node or a pool) fills orders from two pools, Pool B and Pool C, and for some reason (a shorter timeout interval) Pool B is prone to repeating its unfilled orders more frequently than Pool C does. (The **"squeaky wheel effect"**.) This will result in Node A, once the supply flow resumes, sending a disproportionate amount of product to Pool B (as compared to Pool C).

To alleviate this problem, we pair order timeout with **order cancelation**. Whenever a pool decides that an order it has placed with supplier A has time out, it sends a cancelation message, which includes the original order ID, to node A. When node A receives the cancelation message, it reviews its order book, and if it still contains the specified order, the order is deleted. This prevents the supplier from filling the order that the consumer is not waiting for any more.

## Remaining problems

The order timeout plus order cancelation tandem prevents the MTS mechanism from being permanently disabled by transient disruptions, and in many cases it achieves this without causing the opposite problem, that of over-delivering due to duplicated orders. However, the over-production and over-delivering problem is still possible in certain situations.

Specifically, the over-delivery problem does not appear if the an order timed out because the original order message was lost, or the shipment was lost. The order cancelation is very effective in preventing over-delivery if the downstream pool is canceling an order from an upstream pool that wasn't sending stuff because it was empty.

However, the order cancelation mechanism does not protect the system from over-production and over-delivery when, for example, the DC orders from MedTech, the early stages of the production stage (RM supplier etc) do their part, but the flow of product gets stuck at some later stage (e.g. production halt at the packaging node). In this case, the order cancelation will have no effect, since the places to which the order was re-broadcast by MedTech (such as the RM supplier) have already completed their work, and their semi-finished products (e.g. unfinished devices) are already sitting somewhere in the middle of the chain (e.g. waiting for Packaging node to restart). The upstream node (such as RM suppier does not know what has happened with its product); so it will do nothing about the cancelation message, and, when the repeated order arrives, it will produce a batch of RM again.

I suppose that in real life, in well-organized supply chains, a mechanism exists to "troubleshoot" unfilled orders and find out the "breaking point" or the "bottlneck" that holds things up. So it may be feasible to find out that a given order was (e.g.) broadcast to the RM supplier, the PM supplier, and the DS production unit, what was done by them, and what later happened with the products involved. The MedTech company (with the cooperation of any CMOs involved) could then carry out an appropriate fine-grained cancelation/reordering combination, instructing only some nodes to redo their work. This, of course, is not done in the SC2 app, because I believe that the SC2 model does not assume that units of the supply have access to "wide knowledge" about other units.