Author: Roberto Rossi Email: robros@gmail.com Website: https://gwr3n.github.io

Roberto Rossi, *Inventory Analytics*. Cambridge, UK: Open Book Publishers, 2021, https://doi.org/10.11647/OBP.0252

Please direct all enquiries to the author.

ISBN Paperback: 978-1-80064-176-1
ISBN Digital (PDF): 978-1-80064-177-8
DOI: 10.11647/OBP.0252

Multi-echelon Inventory Systems

https://doi.org/10.11647/OBP.0252.05

Introduction

In this chapter, we briefly survey key aspects related to the control of multi-echelon inventory systems. These are systems in which multiple interconnected installations are present and must be controlled jointly. We first introduce serial systems and associated optimal control strategies; we show how to simulate these systems, and how to compute optimal policy parameters. Finally, we survey other possible multi-echelon inventory systems: assembly systems, distribution systems, and general systems.

Topics

Serial systems

We shall consider a simple serial inventory system comprising two installations: a warehouse W, and a retailer R (Fig. 110). This setup was first investigated in [Clark and Scarf, 1960].

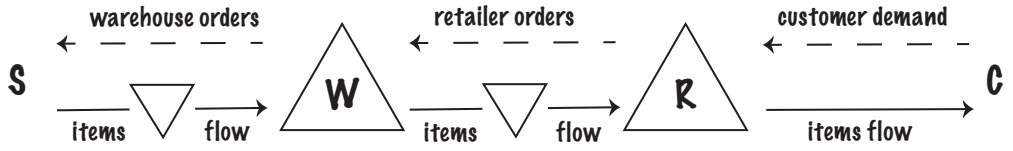

Fig. 110 A serial inventory system comprising two installations: a warehouse and a retailer; physical flows and information flows are represented via solid and dashed lines, respectively.

The system operates in a periodic review setting. We assume installation R faces a customer demand that is stochastic, Poisson distributed with rate λ in each period, and independent across periods. Demand that cannot be met immediately from stock at installation R is backordered. Installation R replenishes from installation W; while installation W replenishes from an infinite outside supply S. Lead times are deterministic and equal to a given (integer) number of periods. There are standard holding costs at installations W and R, and a backorder/penalty cost at installation R. For the time being, we will assume that no ordering/setup costs are present.

At the beginning of each period, the order of events is as follows:

- installation W orders;
- the period delivery from the outside supplier S arrives at W;
- installation R orders from installation W;
- the period delivery from installation W arrives at installation R;
- the stochastic customer demand at installation R is realised;
- evaluation of holding and shortage costs.

We introduce the following notation:

l_i	lead time at installation $i \in \{W, R\}$;
h_i	holding cost per period at installation $i \in \{W, R\}$;
b	backorder/penalty cost per period (only charged at installation R);
I_i	installation stock inventory level at installation $i \in \{W, R\}$ just before period demand;
d_n	n-period stochastic demand, i.e. Poisson$(n\lambda)$.

Our aim is to minimise the expected total holding and backordering cost per period.

Note that we only charge holding costs for stock on hand at the installation, i.e. we do not charge holding costs on in-transit stock between installation W and installation R, as it can be proved that this cost ($h_W \lambda l_W$) is not affected by the control policy.

Simulating a serial system

We will simulate a simple serial inventory system comprising two installations via DES. We first introduce our usual DES engine class.

```
import matplotlib.pyplot as plt, numpy as np, pandas as pd
from queue import PriorityQueue
from collections import defaultdict
from typing import List

class EventWrapper():
   def __init__(self, event):
      self.event = event

   def __lt__(self, other):
      return self.event.priority < other.event.priority

class DES():
   def __init__(self, end):
      self.events, self.end, self.time = PriorityQueue() , end, 0

   def start(self):
      while True:
         event = self.events.get()
         self.time = event[0]
         if self.time <= self.end:
            event[1].event.end()
         else:
            break

   def schedule(self, event: EventWrapper, time_lag: int):
      self.events.put((self.time + time_lag, event))
```

Next, we model our two installations: the warehouse W,

```
class Warehouse:
   def __init__(self, inventory_level, holding_cost, lead_time):
      self.i, self.h, self.lead_time = inventory_level, holding_cost, lead_time
      self.o = 0 # outstanding_orders
      self.period_costs = defaultdict(int) # a dictionary recording cost in each
           period

   def receive_order(self, Q, time):
      self.review_inventory(time)
      self.i, self.o = self.i + Q, self.o - Q
      self.review_inventory(time)

   def order(self, Q, time):
      self.review_inventory(time)
      self.o += Q
      self.review_inventory(time)

   def on_hand_inventory(self):
      return max(0,self.i)

   def backorders(self):
      return max(0,-self.i)

   def issue(self, demand, time):
      self.review_inventory(time)
      self.i = self.i-demand

   def inventory_position(self):
      return self.o+self.i

   def review_inventory(self, time):
      try:
         self.levels.append([time, self.i])
         self.on_hand.append([time, self.on_hand_inventory()])
         self.positions.append([time, self.inventory_position()])
      except AttributeError:
         self.levels, self.on_hand = [[0, self.i]], [[0, self.on_hand_inventory()]]
         self.positions = [[0, self.inventory_position()]]

   def incur_end_of_period_costs(self, time): # incur holding and penalty costs
      self._incur_holding_cost(time)

   def _incur_holding_cost(self, time): # incur holding cost and store it in a
        dictionary
      self.period_costs[time] += self.on_hand_inventory()*self.h
```

and the retailer R.

```
class Retailer:
   def __init__(self, inventory_level, holding_cost, penalty_cost, lead_time,
         demand_rate):
      self.i, self.h, self.p, self.lead_time, self.demand_rate = inventory_level,
            holding_cost, penalty_cost, lead_time, demand_rate
      self.o = 0 # outstanding_orders
      self.period_costs = defaultdict(int) # a dictionary recording cost in each
            period

   def receive_order(self, Q, time):
      self.review_inventory(time)
      self.i, self.o = self.i + Q, self.o - Q
      self.review_inventory(time)

   def order(self, Q, time):
      self.review_inventory(time)
      self.o += Q
      self.review_inventory(time)

   def on_hand_inventory(self):
      return max(0,self.i)

   def backorders(self):
      return max(0,-self.i)

   def issue(self, demand, time):
      self.review_inventory(time)
      self.i = self.i-demand

   def inventory_position(self):
      return self.o+self.i

   def review_inventory(self, time):
      try:
         self.levels.append([time, self.i])
         self.on_hand.append([time, self.on_hand_inventory()])
         self.positions.append([time, self.inventory_position()])
      except AttributeError:
         self.levels, self.on_hand = [[0, self.i]], [[0, self.on_hand_inventory()]]
         self.positions = [[0, self.inventory_position()]]

   def incur_end_of_period_costs(self, time): # incur holding and penalty costs
      self._incur_holding_cost(time)
      self._incur_penalty_cost(time)

   def _incur_holding_cost(self, time): # incur holding cost and store it in a
         dictionary
      self.period_costs[time] += self.on_hand_inventory()*self.h

   def _incur_penalty_cost(self, time): # incur penalty cost and store it in a
         dictionary
      self.period_costs[time] += self.backorders()*self.p
```

Finally, we model the relevant events, to which we assign priorities in line with the order previously illustrated.

```
class CustomerDemand:
   def __init__(self, des: DES, demand_rate: float, retailer: Retailer):
      self.d = demand_rate # the demand rate per period
      self.r = retailer # the retailer
      self.des = des # the Discrete Event Simulation engine
      self.priority = 5 # denotes a low priority

   def end(self):
      self.r.issue(1, self.des.time)
      self.des.schedule(EventWrapper(self), np.random.exponential(1/self.d)) #
            schedule another demand

class EndOfPeriod:
   def __init__(self, des: DES, warehouse: Warehouse, retailer: Retailer):
      self.w = warehouse # the warehouse
      self.r = retailer # the retailer
      self.des = des # the Discrete Event Simulation engine
      self.priority = 0 # denotes a high priority

   def end(self):
      self.w.incur_end_of_period_costs(self.des.time-1)
      self.r.incur_end_of_period_costs(self.des.time-1)
      self.des.schedule(EventWrapper(EndOfPeriod(self.des, self.w, self.r)), 1)
```

```
class OrderUpTo_Warehouse:
   def __init__(self, des: DES, S: float, warehouse: Warehouse, retailer: Retailer):
      self.S = S # the order-up-to-position
      self.w = warehouse # the warehouse
      self.r = retailer # the retailer
      self.des = des # the Discrete Event Simulation engine
      self.priority = 1 # denotes a medium priority

   def end(self):
      Q = self.S - self.w.inventory_position()
      self.w.order(Q, self.des.time)
      self.des.schedule(EventWrapper(ReceiveOrder_Warehouse(self.des, Q, self.w,
           self.r)), self.w.lead_time)

class OrderUpTo_Retailer:
   def __init__(self, des: DES, S: float, warehouse: Warehouse, retailer: Retailer):
      self.S = S # the order-up-to-position
      self.w = warehouse # the warehouse
      self.r = retailer # the retailer
      self.des = des # the Discrete Event Simulation engine
      self.priority = 2 # denotes a medium priority

   def end(self):
      Q = self.S - self.r.inventory_position()
      self.r.order(Q, self.des.time)
      Q_available = min(Q, self.w.on_hand_inventory())
      self.w.issue(Q, self.des.time)
      self.des.schedule(EventWrapper(ReceiveOrder_Retailer(self.des, Q_available,
           self.r)), self.r.lead_time)

class ReceiveOrder_Warehouse:
   def __init__(self, des: DES, Q: float, warehouse: Warehouse, retailer: Retailer):
      self.Q = Q # the order quantity
      self.w = warehouse # the warehouse
      self.r = retailer # the retailer
      self.des = des # the Discrete Event Simulation engine
      self.priority = 3 # denotes a medium priority

   def end(self):
      backorders = self.w.backorders()
      self.w.receive_order(self.Q, self.des.time)
      if backorders > 0:
         q = min(self.Q, backorders)
         self.des.schedule(EventWrapper(ReceiveOrder_Retailer(self.des, q, self.r)),
              self.r.lead_time)

class ReceiveOrder_Retailer:
   def __init__(self, des: DES, Q: float, retailer: Retailer):
      self.Q = Q # the order quantity
      self.r = retailer # the retailer
      self.des = des # the Discrete Event Simulation engine
      self.priority = 4 # denotes a medium priority

   def end(self):
      self.r.receive_order(self.Q, self.des.time)
```

The `simulate` method runs the DES (Fig. 111). `S_r` represents the order up to position for the retailer, and `S_w` represents the order up to position for the warehouse, `N` is the number of periods to be simulated.

```
def simulate(retailer, S_r, warehouse, S_w, N, plot):
   np.random.seed(1234)
   r, w = Retailer(**retailer), Warehouse(**warehouse)

   des = DES(N)
   d = CustomerDemand(des, r.demand_rate, r)
   des.schedule(EventWrapper(d), 0) # schedule a demand immediately

   o_r = OrderUpTo_Retailer(des, S_r, w, r)
   o_w = OrderUpTo_Warehouse(des, S_w, w, r)
   for t in range(N):
      des.schedule(EventWrapper(o_r), t) # schedule orders
      des.schedule(EventWrapper(o_w), t) # schedule orders
   des.schedule(EventWrapper(EndOfPeriod(des, w, r)), 1) # schedule EndOfPeriod at
        the end of the first period
   des.start()

   tc = sum([w.period_costs[e] for e in w.period_costs]) + sum([r.period_costs[e]
        for e in r.period_costs])
   return tc/N
```

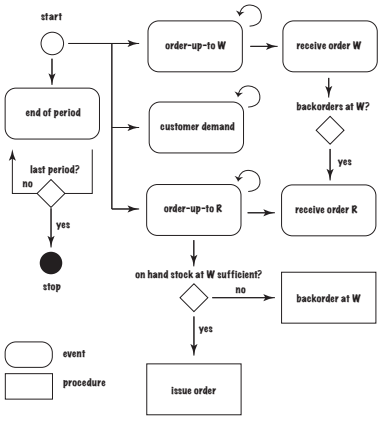

Fig. 111 A serial inventory system comprising two installations: DES flow diagram.

Example 39. *We consider the following instance: retailer holding cost $h_r = 1.5$, retailer backorder/penalty cost $b = 10$, retailer order leadtime $l_r = 5$, customer demand follows a Poisson distribution with rate $\lambda = 10$, warehouse holding cost $h_w = 1$, warehouse leadtime $l_w = 5$. For the sake of illustration we will set $S_r = 74$ and $S_w = 59$.*

The previous example can be simulated as follows.

```
N = 10000 # planning horizon length
S_r, S_w = 74, 59
retailer = {"inventory_level": S_r, "holding_cost": 1.5, "penalty_cost": 10,
      "lead_time": 5, "demand_rate": 10}
warehouse = {"inventory_level": S_w, "holding_cost": 1, "lead_time": 5}
print("Avg cost per period: "+ '%.2f' % simulate(retailer, S_r, warehouse, S_w, N))
```

The simulated average cost per period is 26.11. The behaviour of the inventory level for periods $5, \ldots, 14$ is shown in Fig. 112.

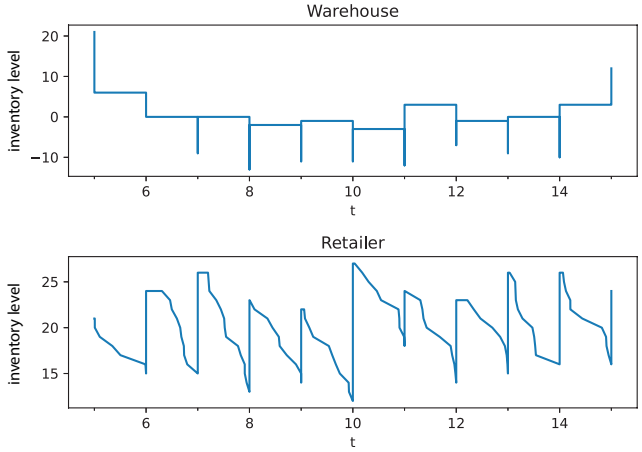

Fig. 112 A serial inventory system comprising two installations: behaviour of the inventory level at installations W and R.

Order-up-to-positions at the retailer and warehouse have been set arbitrarily to $S_r = 74$ and $S_w = 59$, respectively. However, a manger would ideally like to set optimal values for S_r and S_w. A naïve approach to computing optimal values for S_r and S_w is the brute force approach, which explores all possible integer[62] combinations of S_r and S_w (Fig. 113).

[62] Since demand can only take integer values, we can restrict the search to integer combinations.

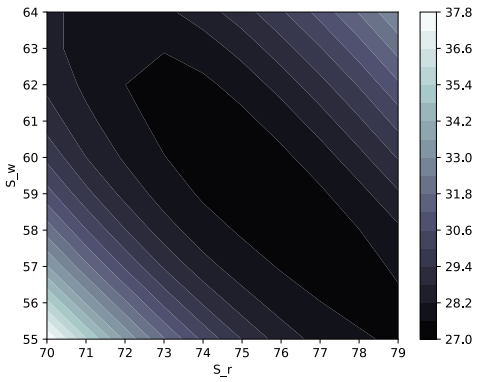

Fig. 113 A serial inventory system comprising two installations: average cost per period for different combination of S_r and S_w; the chosen combination $S_r = 74$ and $S_w = 59$ appears to minimise the expected total cost per period, or at least to be a solution close to the optimal one. Observe that the cost function appears to be convex.

Computing optimal base-stock policy parameters

Instead of relying on the brute force approach, we here illustrate an exact approach[63] for computing optimal base-stock policy parameters for serial inventory systems under stationary demand and an infinite planning horizon. Without loss of generality, we shall focus on the two-installations case; these results are easily extended to an arbitrary number of installations.

[63] Andrew J. Clark and Herbert Scarf. Optimal policies for a multi-echelon inventory problem. *Management Science*, 6(4):475–490, 1960.

Let O_i denote the **outstanding orders**[64] at installation i in any given period; and recall the following definitions.

[64] Outstanding orders are orders that have been issued but not yet received due to the delivery lead time.

The **on hand inventory** at installation i is the positive part of the installation stock inventory level I_i, i.e. $\max\{I_i, 0\} \triangleq [I_i]^+$.

The **installation stock inventory position** Y_i is defined as follows: $Y_i \triangleq O_i + I_i$.

In their exact approach, Clark and Scarf leverage the concept of **echelon stock** (Fig. 114) to compute optimal base-stock policy parameters for serial inventory systems.

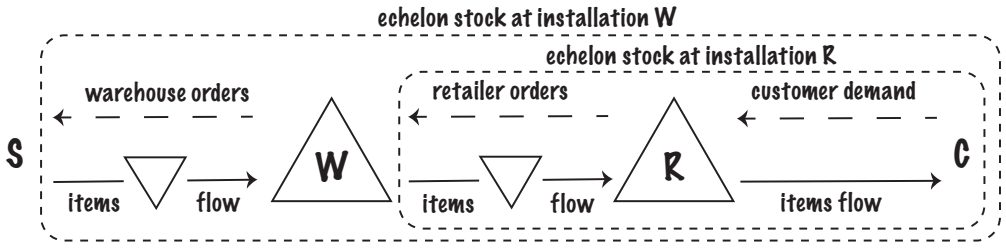

Fig. 114 A serial inventory system comprising two installations: echelon stock at installations W and R.

In essence, *an echelon stock tracks not just the installation stock, but also the downstream stock of an item.*

For an installation i that serves other downstream installations,[65] the **echelon stock inventory position** Y_i^e is defined as follows.

[65] Such as our installation W, the warehouse.

Definition 26. *The echelon stock inventory position is the installation stock inventory position plus the sum of the installation inventory positions at all downstream installations.*

Hence, e.g. $Y_W^e \triangleq Y_W + Y_R$.

Corollary 6. *For installations that do not have further downstream installations they serve,*[66] *the echelon stock inventory position is the same as the installation stock inventory position.*

[66] Such as our installation R, the retailer.

Finally, the **realised echelon stock inventory position** is defined as follows.

Definition 27. *The realised echelon stock inventory position y_i^e at installation i is equal to the echelon stock inventory position Y_i^e minus those outstanding orders, which are backordered at the installation upstream.*[67]

[67] Observe that if the retailer issues an order of size Q at time t, receipt of such a quantity at time $t + l_R$ is not guaranteed, because the warehouse may have backordered this request due to lack of sufficient on hand stock.

Clearly, since the warehouse W replenishes from an infinite supply S, its realised echelon stock inventory position is equal to its echelon stock inventory position. While at the retailer R, these two quantities may differ.

Recall that our aim is to *minimise the expected total holding and backordering cost per period*; and that we only charge holding costs for stock on hand at the installation, i.e. we do not charge holding costs on in-transit stock between installation W and installation R.

We define the **echelon holding cost** $e_i = h_i - h_{i-1}$ at installation i, as the holding cost on the value added when going from installation $i-1$ to installation i, where $i-1$ denotes the installation *upstream* to installation i. Clearly, $e_W = h_W$ since the warehouse has no upstream installation; while $e_R = h_R - h_W$.

First, we should bear in mind that, since the system is stationary, all periods are equal for the purpose of computing the expected total cost per period.

For installation W we consider an order in period t, and costs at the end of period $t + l_W$. For installation R, we consider an order in period $t + l_W$, and costs at the end of period $t + l_W + l_R$.

Lemma 55. *Consider period $t + l_W$, the installation stock inventory on hand at W, after serving the order received from installation R, is*

$$[I_W]^+ = I_W^e - y_R^e$$

and remains stable throughout the rest of the period.

Lemma 56. *The average holding costs at installation W in period $t + l_W$ is*

$$\begin{aligned} h_W E[I_W^e - y_R^e] &= h_W E[y_W^e - d_{l_W+1} - y_R^e] \\ &= h_W(y_W^e - (l_W+1)\lambda) - h_W y_R^e \end{aligned} \tag{40}$$

Lemma 57. *The average holding and backordering costs at installation R in period $t + l_W + l_R$ are*

$$\begin{aligned} &h_R E[y_R^e - d_{l_R+1}]^+ + bE[d_{l_R+1} - y_R^e]^+ \\ &= h_R(y_R^e - (l_R+1)\lambda) + (h_R+b)E[d_{l_R+1} - y_R^e]^+ \end{aligned} \tag{41}$$

Observe that the expected total period costs are a function of y_R^e and y_W^e.

We now reallocate term $-h_W y_R^e$ from Eq. 40 to Eq. 41. By using the fact that $h_R - h_W = e_R$, we obtain

$$C_W(y_W^e) \triangleq h_W(y_W^e - (l_W+1)\lambda) \tag{42}$$

$$C_R(y_R^e) \triangleq e_R y_R^e - h_R(l_R+1)\lambda + (h_R+b)\mathrm{E}[d_{l_R+1} - y_R^e]^+ \tag{43}$$

where Eq. 42 is independent of y_R^e, and Eq. 43 is independent of y_W^e.

Observe that $\mathcal{L}(y) \triangleq \mathrm{E}[d_{l_R+1} - y]^+$ is the first order loss function, which is convex; and since $\mathcal{L}'(y) = F(y) - 1$, where F is the cumulative distribution of d_{l_R+1} [Rossi et al., 2014b, Lemma 1], then the optimal $\hat{y}_R^e$ can be easily obtained from the first order condition

$$\frac{\mathrm{d}C_R(y)}{\mathrm{d}y} = e_R + (h_R+b)(F(y) - 1) = 0$$

that is, by solving

$$F(y) = \frac{h_W + b}{h_R + b}.$$

From the definition of y_W^e and y_R^e, it follows that

$$y_R^e \leq I_W^e = y_W^e - d_{l_W+1}.$$

If $y_W^e - d_{l_W+1} \geq \hat{y}_R^e$, then $\hat{y}_R^e$ is the value that minimises C_R. However, if $y_W^e - d_{l_W+1} < \hat{y}_R^e$, then $\hat{y}_R^e$ cannot be attained, and the best possible value that minimises $C_R(y)$ is $y = y_W^e - d_{l_W+1}$, due to convexity of C_R. This means that the optimal policy is an (echelon stock) base-stock policy with (echelon) order-up-to-position $S_R^e = \hat{y}_R^e$. Furthermore, this optimal policy at R is independent of y_W^e.

Finally, we determine the optimal policy at W. We consider the expected total cost C for the system, when an optimal policy is implemented at R:

$$C(y_W^e) \triangleq C_W(y_W^e) + C_R(\hat{y}_R^e) + \underbrace{\sum_{y_W^e - \hat{y}_R^e}^{\infty} (C_R(y_W^e - u) - C_R(\hat{y}_R^e))\, f(u)}_{\text{shortage costs at installation } W},$$

where f is the probability mass function of d_{l_W+1}, that is a Poisson distributed random variable with rate $(l_W + 1)\lambda$. The last term of C can be interpreted as the shortage costs at installation W induced by its inability to deliver on time to installation R.

Function C is convex, and therefore can be easily minimised. Let $\hat{y}_W^e$ be the global minimum of C, since supplier S has infinite capacity, the optimal policy at W is an (echelon stock) base-stock policy with (echelon) order-up-to-position $S_W^e = \hat{y}_W^e$.

Finally, observe that it is easy to switch from an echelon to an installation base-stock policy, by bearing in mind that the installation order-up-to-position $S_W = S_W^e - S_R^e$.

We next present a Python implementation of these results.

```
import math, matplotlib.pyplot as plt
from scipy.stats import poisson

# cost at the retailer
def C_R(y, e_R, h_R, b, L_R, demand_rate):
   M = round(6*math.sqrt((L_R+1)*demand_rate)+(L_R+1)*demand_rate) # safe upper
         bound: 6 sigma
   return y*e_R-h_R*(L_R+1)*demand_rate+(h_R+b)*sum([(d-y)*poisson.pmf(d,
         (L_R+1)*demand_rate) for d in range(y,M)])

# retailer (echelon) order-up-to-position
def compute_y_R(h_W, h_R, b, L_R, demand_rate):
   return poisson.ppf((h_W + b)/(h_R + b), (L_R+1)*demand_rate)

# expected total cost C for the serial system, when an optimal policy is
      implemented at R
def C(y, e_R, h_R, b, L_R, h_W, L_W, demand_rate):
   y_R = int(compute_y_R(h_W, h_R, b, L_R, demand_rate))
   CW = h_W*(y - (L_W+1)*demand_rate)
   CR = C_R(y_R, e_R, h_R, b, L_R, demand_rate)
   M = round(6*math.sqrt((L_W+1)*demand_rate)+(L_W+1)*demand_rate)
   s = sum([(C_R(y-d, e_R, h_R, b, L_R, demand_rate) - CR)*poisson.pmf(d,
         (L_W+1)*demand_rate) for d in range(y-y_R,M)])
   return CW + CR + s

# warehouse (echelon) order-up-to-position
def compute_y_W(e_R, h_R, b, L_R, h_W, L_W, demand_rate, initial_value):
   y, c = initial_value, C(initial_value, e_R, h_R, b, L_R, h_W, L_W, demand_rate)
   c_new = C(y + 1, e_R, h_R, b, L_R, h_W, L_W, demand_rate)
   while c_new < c:
      c = c_new
      y = y + 1
      c_new = C(y + 1, e_R, h_R, b, L_R, h_W, L_W, demand_rate)
   return y
```

We consider once more the instance in Example 39. The optimal solution can be obtained as follows.

```
retailer = {"holding_cost": 1.5, "penalty_cost": 10, "lead_time": 5, "demand_rate":
      10}
warehouse = {"holding_cost": 1, "lead_time": 5}

h_W, h_R = warehouse["holding_cost"], retailer["holding_cost"]
e_W = h_W
e_R = h_R - e_W
b, demand_rate = retailer["penalty_cost"], retailer["demand_rate"]
L_R, L_W = retailer["lead_time"], warehouse["lead_time"]

initial_value = 100
ye_R = compute_y_R(h_W, h_R, b, L_R, demand_rate)
ye_W = compute_y_W(e_R, h_R, b, L_R, h_W, L_W, demand_rate, initial_value)
print("y^e_R="+str(ye_R))
print("y^e_W="+str(ye_W))
print("y_W="+str(ye_W-ye_R))
print("C(y^e_W)="+str(C(ye_W, e_R, h_R, b, L_R, h_W, L_W, demand_rate)))
```

The solution is $\hat{y}_R^e = S_R^e = S_R = 74$, $\hat{y}_W^e = S_W^e = 133$, $\hat{y}_W = S_W^e - S_R^e = S_w = 59$; and has a cost $C(\hat{y}_W^e) = 26.36$. This is indeed the same solution we previously considered. In Fig. 115 we plot $C_R(y)$; in Fig. 116 we plot the expected total cost $C(y)$ for the system.

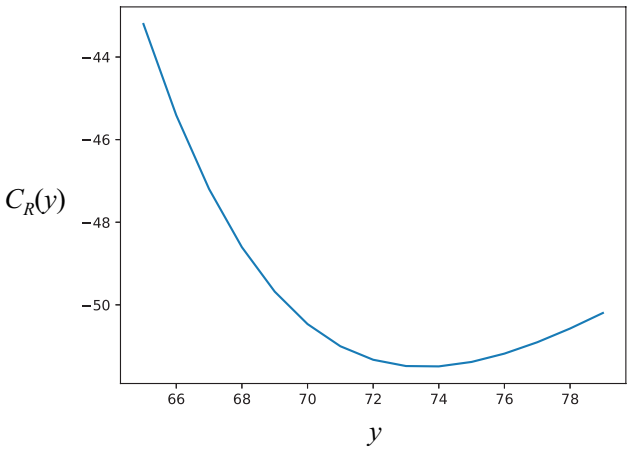

Fig. 115 A serial inventory system comprising two installations: $C_R(y)$.

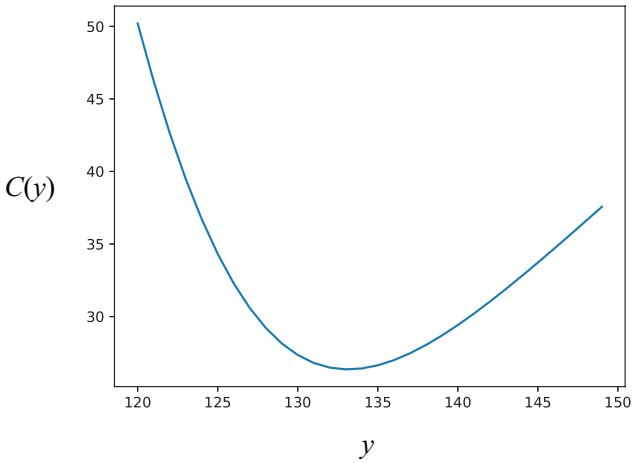

Fig. 116 A serial inventory system comprising two installations: $C(y)$.

Assembly systems

An assembly system is a multi-echelon production line in which end products are manufactured from more basic components. An example of an assembly system is shown in Fig. 117.

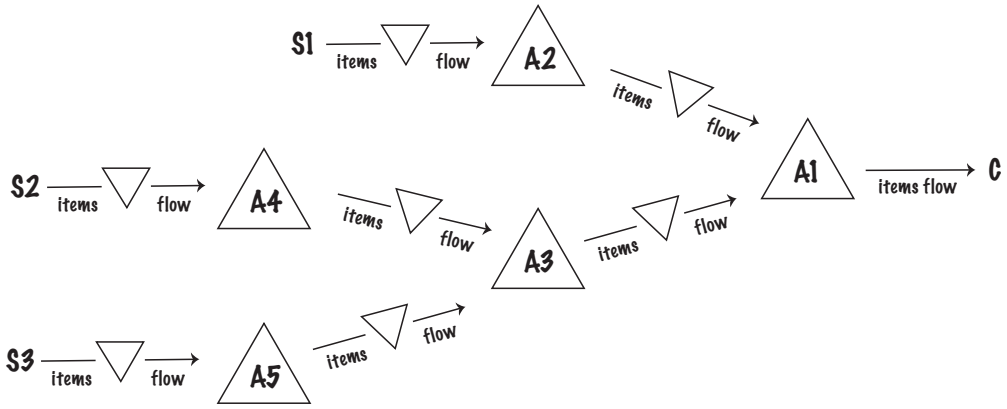

Fig. 117 An assembly system.

Every assembly system can be transformed into an equivalent serial system,[68] therefore dealing with assembly systems is no more difficult than dealing with its equivalent serial system.

[68] Kaj Rosling. Optimal inventory policies for assembly systems under random demands. *Operations Research*, 37(4):565–579, 1989.

Distribution systems

A distribution system is a multi-echelon inventory system in which a given product is distributed from a supply to a number of downstream installations, in order to serve some sources of demand. An example of a distribution system is shown in Fig. 118.

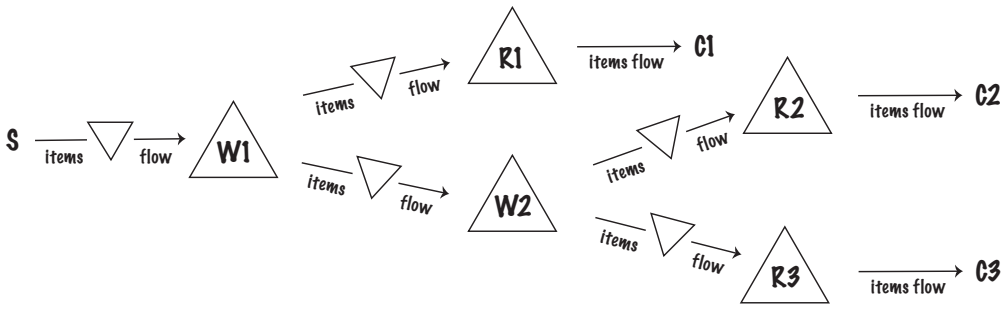

Fig. 118 A distribution system.

Distribution systems are difficult to control. The difficulty stems from the fact that, when inventory is scarce, upstream installations must decide how to allocate this scarce resource optimally to serve demand coming from their downstream installations. Possible stock allocation strategies may include: first-come first-serve, stock balancing, or priority-based. The structure of the optimal control policy is generally not known for these systems; it is therefore customary to fix a control policy (e.g. base-stock policy) and an allocation strategy (e.g. first-come first-serve), and then compute optimal or near-optimal policy parameters under these assumptions.

A well-known distribution system that has been widely studied

in the literature is the one-warehouse multiple-retailers system, which comprises a single upstream installation (the warehouse) that serves directly several downstream installations (the retailers). It is common to use Clark and Scarf's approach for tackling this system. However, this approach is no longer exact. The underpinning approximation consists in allowing the warehouse to implement negative allocations at its downstream installations. This means the total stock at the retailers can be optimally "reshuffled" between sites at any period. This technique was illustrated in [Clark and Scarf, 1960]. A well-known heuristic approach for this system is the so-called METRIC, which was originally introduced in [Sherbrooke, 1968]. A computationally expensive exact approach, the "projection algorithm," to tackle the one-warehouse multiple-retailers system was introduced in [Axsäter, 1990].

The one-warehouse multiple-retailers system

General systems

General systems take the form illustrated in Fig. 119.

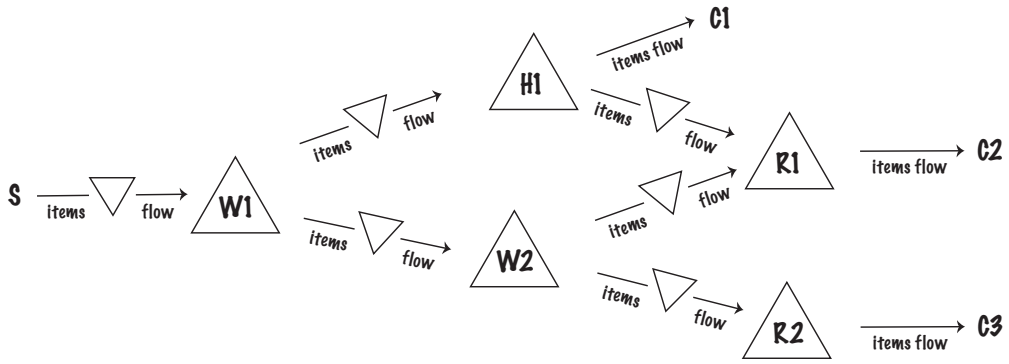

Fig. 119 A general multi-echelon system, in which installations can have multiple successors as well as predecessors.

Due to their very general structure, it is difficult to derive properties and/or insights on the nature and structure of the optimal control policy for such systems.

For instance, consider the following difficulty: in production it is common to assemble multiple components into a final or intermediate product. In the case of serial and assembly systems, this complication is only apparent. In these systems, a component can only be part of a given item. Therefore we can always redefine the unit of measure: e.g. if a given item contains two units of a given component, we can define two units of this component to be the new unit. However, this is not possible in general multi-echelon system such as the one in Fig. 119. To see this, consider installation $H1$, where H denotes a "hybrid" installation that can serve as a retailer but also as a warehouse. $H1$ can sell components directly to customers $C1$, or it may supply these components to $R1$. It may happen that the product manufactured by $R1$ contains two units of the component supplied by $H1$. In this case, it is clear it is not possible to avoid the difficulty by redefining the unit.