

# UDPP analysis

Lorenzo Castelli  
Andrea Gasparin

*Università degli studi di Trieste - Italy*

22/07/2020

## 1 Overview

In modern *ATM* the flight delay impact caused by a **Constraint Capacity Situation** (CCS) represents an important problem, which has been widely studied in literature. In particular, many attempts have been made to develop new *Collaborative Decision Making* (CDM) frameworks in order to help the *Airspace Users* (AUs) to reduce the additional costs caused by delays. Toward this aim, from 2004 in the context of SESAR programme (Single European Sky ATM Research), promoted by the European Union, the development of new mechanisms to favour the active role of the AUs in the slot allocation process under CCSs has been tried. In particular, it was introduced the possibility for the AUs to indicate their flight priorities within a framework called **User Driven Prioritisation Process** (UDPP). One of the first important formulations of a UDPP algorithm, has been shown by [1]. In the mechanism analysed, AUs priorities are used to determine a **flight delay apportionment** (FDA), a feature which allows an AU to distribute the amount of delay it must absorb among its flights. Moreover, a feature called **Selective flight protection** (SFP) has been developed. The SFP, allows an AU to gain a certain amount of credits, each time it decides to further increase the delay of one of its flights considered less important; these credits can be then spent, by the same AU, in order to reduce the delay of another of its more strategical flights. The UDPP framework and its features, have been periodically tested in several SESAR exercises, and further improvements have been developed. In its current version, described in **SESAR Solution PJ07.02 SPR-INTEROP/OSD for V2 - Part I** (2019) and that it will be analysed in details in this document, a new formulation of SFP has been developed, without credits, and the FDA has been replaced by the **Flight delay reordering** (FDR). This last feature is based on the concept of slot ownership, meaning that in certain sense a slot belongs to the airline which flight is assigned to, and allows each AU to reorder its flights within its

own slots. In addition, one of the main characteristics of the UDPP, is that before determining a final slot assignment, it provides to the AUs a *what if* scenario, consisting in an approximate description of what they would achieve, applying the FDR and SFP features. This extra feature is a remarkable improvement in terms of flexibility, as it also allows airlines to adjust priorities on the spot, before the eventual commitment. UDPP achieved a wide consensus among the AUs and showed satisfactory experimental results. Before to start the detailed analysis of the mechanism we need to introduce a few concepts to better characterise the possible slot swapping operations:

**Definition 1** *Any change of flight order within the same AU defines an **intra-airline operation**.*

*If an intra-airline operation doesn't effect any other AUs, it will be called an operation of **type I**.*


**Definition 2** *If an intra-airline operation has a positive effect on any other AUs, it will be called an operation of **type C**.*

**Definition 3** *A slot exchange involving multiple airlines defines an **inter-airline operation** and we will refer to it as operation of **type T***


The idea behind the UDPP mechanism is to improve flexibility and reduce delay impact on airlines under a CCS, allowing all airlines to perform operations of type **I** and type **C**. The features introduced with the UDPP concept are:

**Priority values:** an airline can define for each of its flight a label that describes its level of importance. They represent the core of the UDPP concept as they are the values used by the algorithm to decide how to perform the operations

**Margins:** a feature that allows airlines, also to specify a target time window for any of its flight, that will be considered during the execution of the UDPP algorithm

 **SPR Selective flight protection:** an operations of type **C**, or in some cases **I**, that allows an airline to prioritise some of their most important flights in order to reduce as much as possible their delay. A flight chosen to be protected is called **Pflight** (protected flight). In exchange, for each *pflight* the airline must accept to increase the delay of one of its earlier flights.

**FDR Flight delay reordering:** an operation of type **I** that allows an airline to reshuffle the order of its flights based on the priority values given

It is important to remark that under a CCS an airline can freely decide whether to use UDPP or not. If an airline decides to not participate to UDPP, its flights will be rescheduled based on the FPFS algorithm, and UDPP will try to not cause negative effects on them, even though,  we will see, it can't be always guaranteed. On the other hand, as long as UDPP includes operations of

type  $\mathcal{C}$ , they might experience also a positive impact, due to the application of the SFA by some other airlines.

The flow of the UDPP mechanism can be summarised as follow:

- Up to a few hours before the beginning of the regulation, each airline declares its will to participate to the UDPP and submits the list of flights defining their preferences, flights priority values and optionally flights *margins*
- UDPP algorithm is then separately applied for each airline, providing a **local solution** using the **Local Flight Time Assignment** (*local UDPP*). This means that, for each airline the algorithm produces a new schedule taking into account just its priority values and applying the UDPP features just on its flights. The *local UDPP* is a sequential algorithm whose workflow can be described as follow:
  - manage *Pflights*
  - manage flights with *Margins*
  - manage *default* flights; flights for which no explicit priority has been given
  - manage flights with defined priorities (but no margins)
  - manage *suspended* flights; less important flights, that the AU allows to postpone at the end of the Hotspot (also candidate to cancellation)
- Eventually, all *local solutions* are merged to produce the final schedule in a *FPFS* manner, based on their new *local solution* time

## 2 UDPP features

Let's analyse now in details the different features provided by UDPP.

### 2.1 Priority values

As we have seen before, the entire UDPP concept is based on the fact that AUs are allowed to indicate their preferences over their flights in order to obtain a different schedule and so reduce the delay impact caused by a CCS. The *priority values* are the UDPP representation of the AUs preferences, and for each flight they can be set to:

- a **priority number** number. Conventionally in reverse order w.r.t the importance of a flight; e.g. from 1 (highest priority) to 999 (lowest priority). The scale used is a custom parameter which can be decided by each AU
- **P** to protect a flight: very important AU flight(s). If a flight priority is set to **P** this will be called **Pflight**, and other two parameters have to be specified:

- the **Max Delay Protection**: this AU parameter gives the maximum delay acceptable for a *Pflight* according to its schedule time: (e.g. 5mn or 10mn). In our context, for simplicity, in place of *Max Delay Protection* we will refer directly to the corresponding slot
- the **Hotspot Flight Earlier Schedule**. For simplicity, in our context will be interpreted as the slot defined by the expected arrival time *ETA*.
- **L** Lowest priority flight of the AU: equivalent to set the highest value in the AU priority number scale.
- **B** Baseline priority: specifies to keep the baseline delay of the flight as the current target delay.
- **dB** default baseline: instead of assigning to each flight a priority an AU can decide to define a value (either a number or a letter) that will be automatically assigned to a flight when no explicit priority is given.
- **S** to suspend a flight: specifies that the flight will no longer be in the middle of the UDPP measure, up to the AU to take a decision concerning this flight (cancellation, diversion, rerouting ...). It becomes the least important flights of the UDPP measure, and the flight will be allocated after the last not suspended flights of the UDPP measure.

## 2.2 Margins

**Margins** is a not mandatory additional feature, introduced to provide the AU the possibility to express specific time constraints on certain flights. When *Margins* on a flight are declared the algorithm will consider them in order to rearrange flights in such a way that the time constraints defined with the margins will be respected (whether feasible).

Margins on flights can be given by two values:

- **Time not After**: specifies a time by which the flight is requested not to be later than the value indicated.
- **Time not Before**: specifies a time by which the flight is requested not to be earlier than the value indicated.

If *Margins* are defined for *Pflights* they overwrite the *Max Delay Protection* and the *Hotspot Flight Earlier Schedule*. In addition, it is important to remark that flights with defined *Margins*, called **Mflights**, will be considered more import than all others non *Pflights*, meaning that the algorithm will try to reallocate them based on their priority values and their *Margins*, after have managed the *Pflights* **but** before handling all the others.

## 2.3 Selective Flight Protection (SFP)

Flight protection is a feature of *UDPP*, similar to the compression mechanism, that allows airlines to request a delay reduction of some of their flights if they accept an increase of delay for some others. More in details, if a flight priority is set to *P* it is also given a time window defined by the *Max Delay Protection* and the *Hotspot Flight Earlier Schedule*. This time window defines a set of contiguous slots, called **Pobjective**, starting from the **target slot**, the one corresponding to the *Hotspot Flight Earlier Schedule* and the one corresponding to the *Max Delay Protection*. The goal is to assign to the *pflight* a slot within the *Pobjective*. In order to allow this operation and avoid negative effects on other AUs, the AU

- **must** have, a minimum of one slot within or earlier the *Pobjective* of the protected flight, and this slot has not to be currently occupied by another *Pflight* (of the same AU).

In case of multiple *Pflights* this condition doesn't result sufficient. In facts in case of  $n$  *Pflights*, with  $n > 1$ :

- each *Pflight* **must** be matched with an AU slot within or earlier then its *Pobjective* that hasn't been already matched with any of the other *Pflights*.

The idea is that once the slot  $s$  is identified (if possible), then:

- if  $s$  is within the *Pobjective*, the *Pflight* and them assigned to  $s$ . The flight previously in  $s$  will be assigned to a later slot, owned by the AU, in the next stages of the UDPP algorithm.

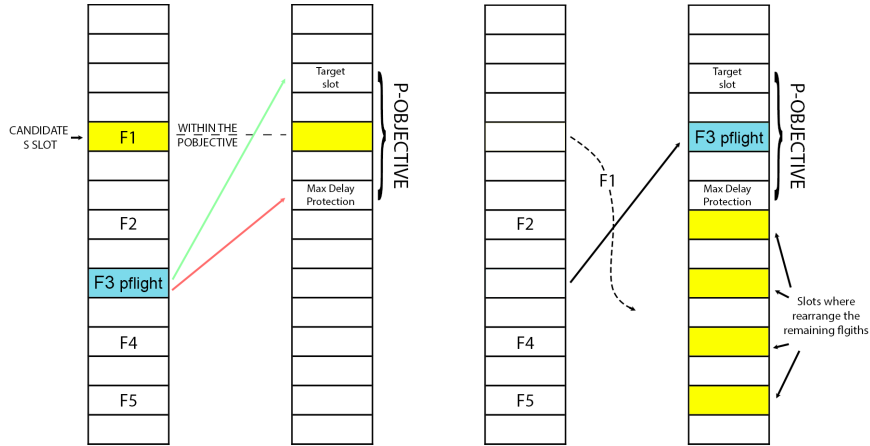


Figure 1: Protection when candidate slot  $s$  is in the *pobjective*

- if  $s$  is earlier than the *target slot*, then the *Pflight* is assigned to the *target slot*.

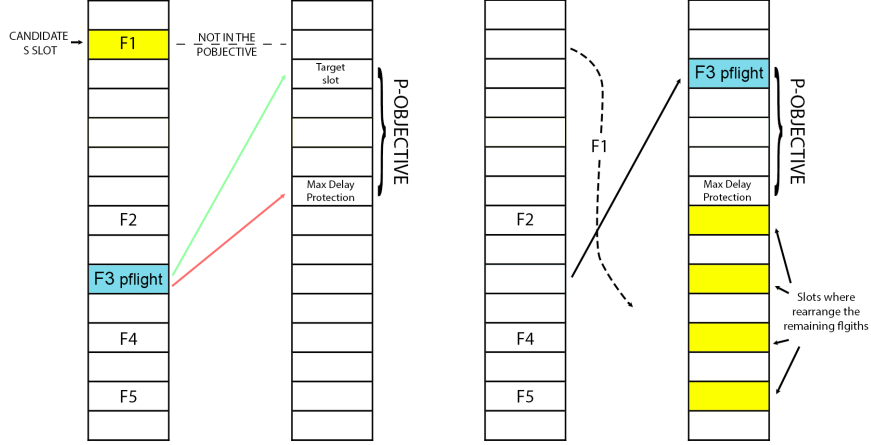


Figure 2: Protection when candidate slot  $s$  is not in the *pobjective*

The way in which the slot  $s$  is identified will be discussed in details later. We can notice for the moment that this operation, is in the first case of type  $\mathcal{I}$ , as it doesn't effect other AUs. In the second case instead, we have that the *target slot* is currently held by another AU: we will see later how, in the final step, the *merge algorithm* manages this kind of circumstance in a *compression* – like manner and how this might potentially produce a positive effect on other AUs, defining an operation of type  $\mathcal{C}$  (but also, in some pathological cases, cause some negative impact)



## 2.4 Flight Delay Reordering (FDR)

This feature consists simply in a reallocation of the flights into the AU available slots, which has been not already assigned by *SFP* to *Pflights*, based on their priority value but clearly respecting the restriction dictated by the *ETA*. The algorithm sorts the flights by priority number and, starting from the highest priority flight to the lowest one, assigns to each flight the first remained free slot compatible with its *ETA*.

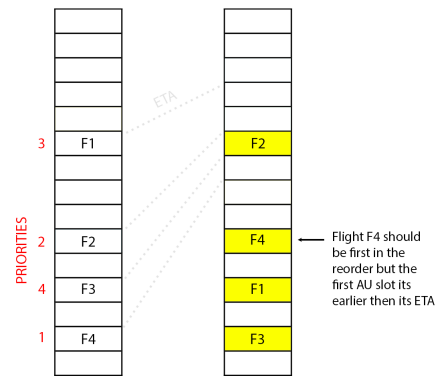


Figure 3: FDR

The same principle is beforehand applied to flights with defined *Margins*, but in this case just also considering the target time window specified by the *Time not Before* and *Time not After*.



### 3 Local UDPP

In this paragraph we will work in the UDPP local environment, so we will assume that just one AU is participating to UDPP and all UDPP features will be applied just on that airline. The schedule obtained in this manner will be the AU's *local solution*. This reasoning can be applied to all airlines actually participating to UDPP and later we will see how, starting from all *local solutions*, the merge algorithm will produce the final UDPP solution which will define the final schedule. In general terms the local UDPP can be seen just as a reorder of the flights, possibly within the slots owned by the AU. The flight protection, which is the first stage of the algorithm, is the only procedure that might assign a flight to a slot owned by another AU, but, as we have seen in 2.3, just if in exchange an earlier slot has been released.

We now remind the definition of the *Local UDPP*:

---

#### Algorithm 1: Local UDPP

---

$slotList$  = set of slots owned by the AU without slots currently  
occupied by flights with priority  $B$

$schedule$  = set of all slots

$Pflights$  = set of flights with priority value  $P$

**ManagePflights**( $Pflights, schedule, slotList$ )

$Mflights$  = set of all  $Mflights$  which are not  $Pflights$

$slotList$  = updated  $slotList$  by *ManagePflights*

**ManageMflights**( $Mflights, slotList$ )

$dBFlights$  = set of all flights with no priority or *Margins*

**ManageDBllights**( $dBFlights, slotList$ )



$Nflights$  = set of flights with priority but not *Margin*

**ManageNflights**( $Nflights, slotList$ )



$Sflights$  = set of flights with priority  $S$

**ManageSflights**( $Sflights, schedule$ )

---

It is important to notice that in the first line of *Algorithm 1* the initial  $slotList$  doesn't include the  $Bflights$ . As long as all consequent functions will operate just on elements within the list, the  $Bflights$  will remain untouched by the *Local UDPP*.



### 3.1 ManagePflights()

The **ManagePflights** consists in the **SFP algorithm** that we introduced in the section 2.3.

We remind that the inputs of this function are:

- *Pflights*: the list of flights with priority  $P$
- *slotList*: set of slots owned by the AU without slots currently occupied by flights with priority  $B$
- *schedule*: the list of all slots within the hotspot

The goal of this function is to assign, whether possible, all *Pflights* to a slot within their *Pobjective* time window. We also remember that when *Margins* are defined, they overwrite the *Max Delay Protection* and the *Hotspot Flight Earlier Schedule*. As long they have the same meaning we can just then use the *Time not After* and the *Time not Before*, and in the case *Margins* are not given, set:



*Time not After* = *Max Delay Protection*

*Time not Before* = *Hotspot Flight Earlier Schedule*



The algorithm can be summarised in this way:

- it sorts the *Pflights* by **Time not After**
- sequentially, for each *Pflight*:
  - it tries to find a slot  $s$  within the *Pobjective*
    - \* if found, it assigns the *Pflight* to  $s$ .  $s$  is then removed from the *slotList*
    - \* otherwise it assigns the *Pflight* to the slot corresponding to the *Time not Before*. In this case the slot allocated to the *Pflight* doesn't belong to the AU. In exchange, the last slot in the *slotList* earlier than the *Time not Before* is removed from the list. See figures 1 and 2
- if after all assignments, the order of the *Pflights* (according to their *Time not After*) is not respected, their slot allocation is reordered.



We will see later that in the case in which a *Pflight* is assigned to a slot of another AU, when the *UDPPmerge* algorithm is executed, this might produce a positive effect on other AUs.

In the next detailed description we will refer to the *Time not After* as  $tnA$  and to the *Time not Before* as  $tnB$ .

---

**Algorithm 2:** ManagePflights (SFP)

---

*Pflights.sort()* by *tnA*; if equal by *Baseline time*

```
for pf in Pflights do
    solutionFound = false
    pfSlot = slot currently occupied by pf
    slot = pf.tnA
    while solutionFound == false and slot.time ≥ pf.tnB do
        if slot in slotList then
            assign pf to slot
            slotList.remove(slot)
            slotList.append(pfSlot)
            slotList.sort() by time
            solutionFound = true
        end
        slot = slot − 1
    end

    if solutionFound == false then
        slot = latest slot in slotList s.t. slot.time < pf.tnB
        pfNewSlot = slot corresponding to pf.tnB
        assign pf to pfNewSlot
        slotList.remove(slot)
        slotList.append(pfSlot)
        slotList.sort() by time
    end
end

if Pflights not sorted by Time not After then
    rearrange the Pflights slots order by Time not After; if equal by
    Baseline time
end
```

---

### 3.2 MenageMflights

The inputs of this function are:

- *Mflights*: the list of the flights with defined *Margins* without the *Pflights*
- *slotList*, that we remind it has been updated by the function *ManagePflights*

The goal here, is to assign a slot to each *Mflight* respecting, where possible, its *Time not After* margin. The algorithm can be summarised in this way:

- it sorts the *Mflights* by priority number, if equal by *Time not after*
- sequentially, for each *Mflight*:
  - it tries to assign the *Mflight* to the latest slot in the *slotList* earlier than the *Mflight Time not after (target slot)*
  - if not possible, it tries to shift up the earlier *Mflights* to free the *target slot*, and then to assign the *Mflight* to it. The shift will be managed by the recursive function *AssignOrShiftEarlier*
  - if not possible, it assigns the *Mflight* to the next slot w.r.t. *target slot*

Reminding that we are referring to the *Time not After* as *tnA* and to the *Time not Before* as *tnB*, the detailed description of the function is:

---

**Algorithm 3:** ManageMflights(*Mflights*, *slotList*)

---

```

Mflights.sort() by priority number, if equal by tnA;

for mf in Mflights do
    mf.targetSlot = latest slot in slotList s.t. slot.time ≤ mf.tnA
    assignmentSuccessful = AssignOrShiftEarlier(mf, slotList)

    if assignmentSuccessful == false then
        assign mf to the next free slot in slotList
    end
end
end

```

---

---

**Algorithm 4:** AssignOrShiftEarlier( $f, slotList$ )

```
if  $f.targetSlot.time < f.tnB$  then
  return( $false$ )
end
if  $f.targetSlot.assigned == false$  then
  assign  $f$  to  $f.targetSlot$ 
   $f.targetSlot.assigned = true$ 
  return( $true$ )
else
   $earlierMflight =$  current assigned flight to  $m.f.targetSlot$ 
   $earlierMflight.targetSlot =$  previous slot in  $slotList$  w.r.t.
   $f.targetSlot$ 
   $assignmentSuccessful = AssignOrShiftEarlier(earlierMflight)$ 

  if  $assignmentSuccessful == true$  then
    assign  $f$  to  $f.TargetSlot$ 
     $f.targetSlot.assigned = true$ 
    return( $true$ )
  else
    return( $false$ )
  end
end
end
```

---

### 3.3 ManageDBflights

The objective of this function is to assign all flights with default priority  $B$ . The inputs of this function are:


- $dBflights$ : the list of all flights with default priority  $B$ , which we remind is the default assignment given to a flight when no explicit priority is given to it
- $slotList$ : updated by the function *ManagePflights*, but including slots assigned by *ManageMflights*

The reason why the  $slotList$  remains untouched by *ManageMflights* is that to handle  $dBflights$ , the function *AssignOrShiftEarlier* (Algorithm 4) is used, so it still might be possible to shift up some  $Mflights$ .

---

**Algorithm 5:** ManageDBflights(*dbFlights*, *slotList*)

---

```
dbFlights.sort() by baseline time of arrival;  
  
for dbf in dbFlights do  
    dbf.TargetSlot = latest slot in slotList s.t.   
    slot.time ≤ mf.baseTimeOfArrival  
  
    assignmentSuccessful = AssignOrShiftEarlier(dbf, slotList)  
  
    if assignmentSuccessful == false then  
        | assign dbf to the next free slot in slotList  
    end  
end
```

---

### 3.4 ManageNflights

The input of this functions are:

- *nFlights*: the list of the flights with priority number with no *Margins* defined, and with no priority of type *P*, *B* or *S*.
- *slotList*: including slots assigned by *mFlights* and *dbFlights*
- *schedule*: all slots in the Hotspot

The idea, is now to allocate all *nFlights* on the remaining free slots according to their priority number. The algorithm sorts the flights by priority number and, starting from the highest priority flight to the lowest one, assigns to each flight the first remained free slot compatible with its *ETA*. In the case where a flight can't be assigned to any of the remained empty slot due to its *ETA*, the function *AssignOrShiftEarlier* is then called, with the target time set to end of the hotspot. This will assign the flight, to the last available slot. The algorithm can be described in details as follow:

---

**Algorithm 6:** ManageNflights

---

$nFlights.sort()$  by priority number, if equal by baseline time;  
 $freeSlots =$  slots in  $slotList$  which are not assigned to any flights yet

```
for  $nf$  in  $nFlights$  do
  if  $freeSlots[0].time \geq nf.eta$  then
    assign  $nf$  to  $freeSlots[0]$ 
     $freeSlots.remove(freeSlot[0])$ 
  else
     $nf.TargetSlot =$  latest slot in  $schedule$ 
     $AssignOrShiftEarlier(nf)$ 
  end
end
```

---

### 3.5 ManageSflights

The handling of the  $sFlights$ , that we recall are those that will no longer be in the middle of the UDPP measure as considered the least important flights of the UDPP measure, will be in the stage simply allocated after the last not suspended flights of the UDPP measure. The actual time allocation will be done by the  $UDPPmerge$  algorithm.

### 3.6 Local UDPP example

Lets now visualise the flow of the *UDPPlocal* with the following:

**Example 1** Here is considered an AU with 5 flights, one protected, no suspended, two with Margins and two for which just priority numbers are given.

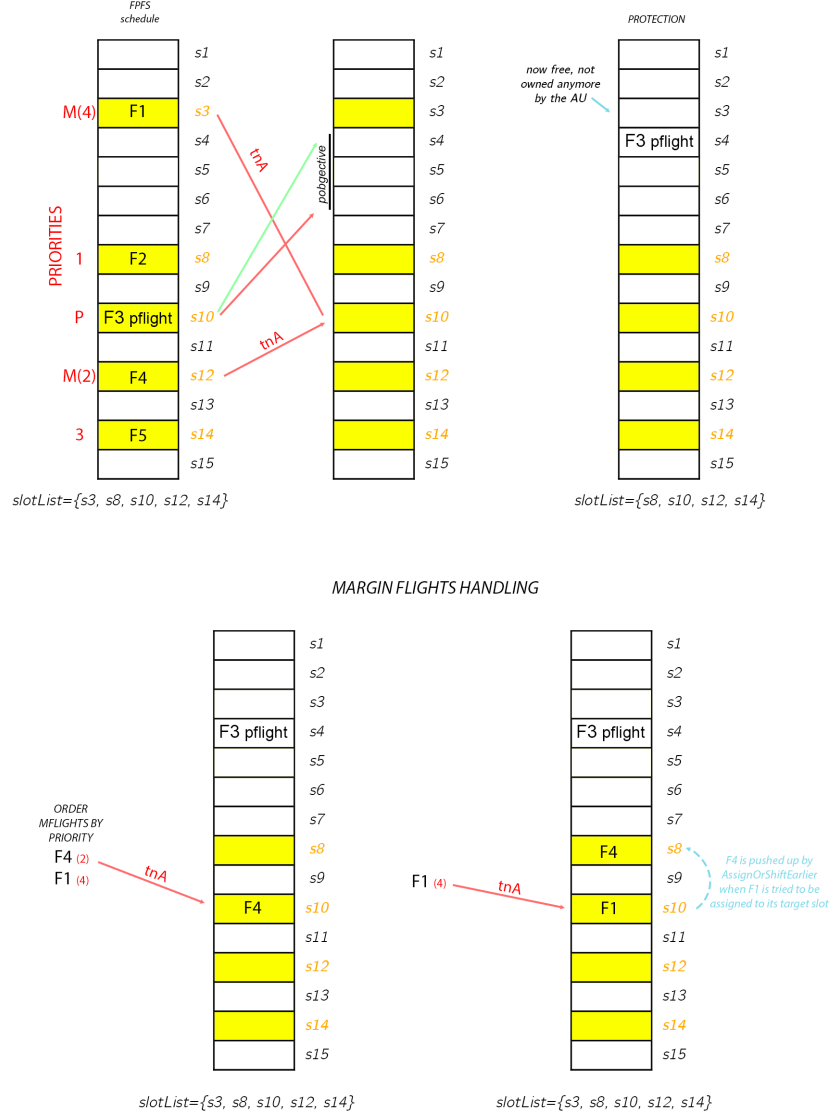


Figure 4: *UDPPlocal* example

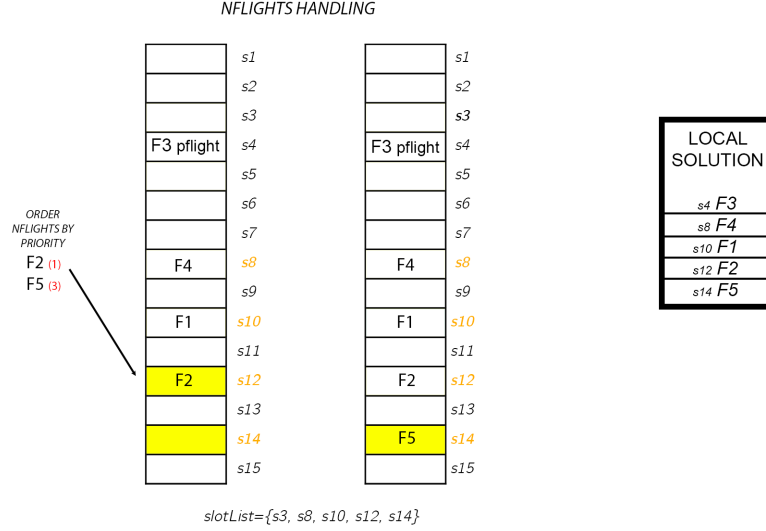


Figure 5: *UDPPlocal* example

## 4 UDPPmerge

This is the last step of the algorithm and the output will be the new final time schedule. The idea is to try to assign all flights to the resources indicated by the AUs *UDPPlocal* algorithm. The local list of the AUs which are not participating to UDPP is by default simply the one generated assigning to each flight its baseline slot.

The procedure can be summarised as follow:

- get all flights which are not *sFlights*
- sort them according to their *UDPPlocal* assignment and in case of conflict according to the baseline allocation
- sequentially, for each flight:
  - assign it to the first available slot not earlier then its ETA
- create an *emptyList* with the slots remained unassigned within the hotspot, and append also empty slots available after the end of the hotspot
- get all *sFlights* and sort them by baseline allocation
- sequentially, for each flight:
  - try to fill the first slot in the *emptyList* that is later or equal to the flight ETA



Before see the details of the algorithm we remind that that if an AU didn't participate to UDPP, for each its flight we set  $flight.newSlot = flight.oldSlot$




---

**Algorithm 7:** UDPPmerge

---

```

flights = list of all non sFlights within the hotspot
flight.sort() by their newSlot parameter, the one obtained with the
  UDPPlocal, if equal by baseline
slotList = all slots in within the hotspot
CASALikeAssignment(flights, slotList)

sFlights = list of all sFlights of all airlines
sFlight.sort() by baseline
freeSlots = slots in slotList which are not assigned to any flights yet
laterFreeSlots = set of size(sFlights) free slot available after the end
  of the hotspot
freeSlots.append(laterFreeSlots)
CASALikeAssignment(sFlights, freeSlots)

```

---



---

**Algorithm 8:** CASALikeAssignment(*flightListm*, *slotList*)

---

```

for f in flightList do
  flightAssigned = false
  while flightAssigned = false do
    i = 0
    if slotList[i].time >= f.ETA then
      assign f to slotList[i]
      slotList.remove(slotList[i])
      flightAssigned = true
    else
      i += 1

```

---

To summarise the flow of the UDPPmerge lets see an:

**Example 2** In an hotspot involving 4 airlines, *a*, *B*, *c*, *D*. Airlines *a* and *c* decide to not use the UDPP, so their local solution will be the one provided by FPFS. *B* and *D* instead, decide to participate to UDPP, and from UDPPlocal they obtain the following local solutions:

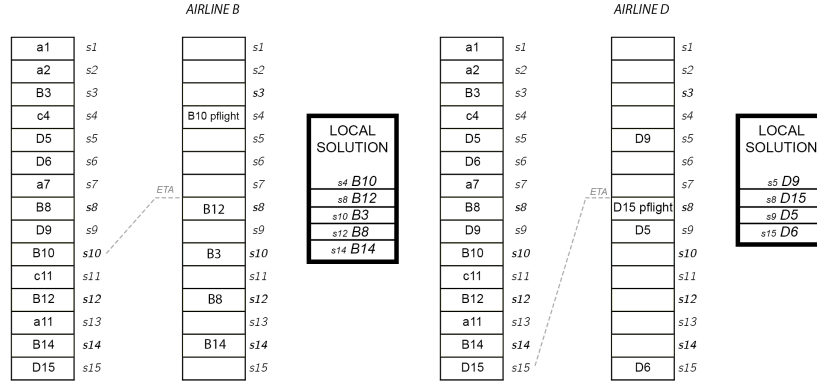


Figure 6: Local solutions for airline B and D

The *UDPPmerge* will first create the *flightList* including all flights within the hotspot, sorted by local solution time and then (in case of conflict) by ETA. After that, in list order, it will assign each flight to the the first free slot; if the latter it's earlier then the flight's ETA, the algorithm will assign it to the first compatible slot. In the example, this is the case of flight B12, which can't be assigned to slot s7:

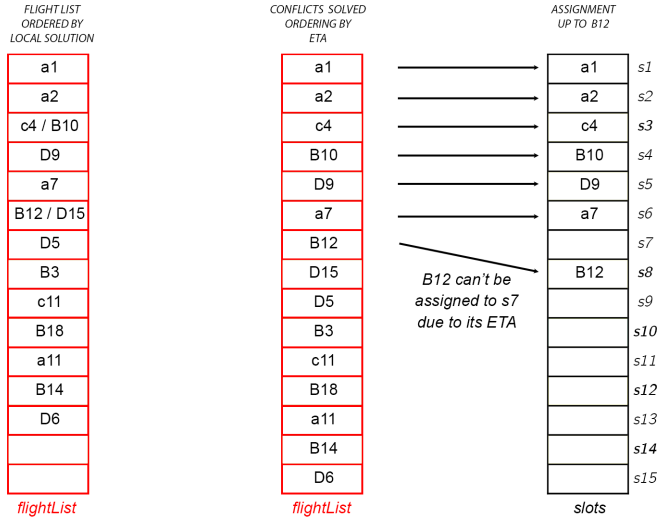


Figure 7: Assignment up to flight B12

s7 is then free, but next flight, D15, can't be assigned to it as, again, it's earlier than its ETA. s8 is occupied by B12, so D15 is assigned to s9. Then, s7 is still the first free slot, but this time it is compatible with next flight D5's ETA:

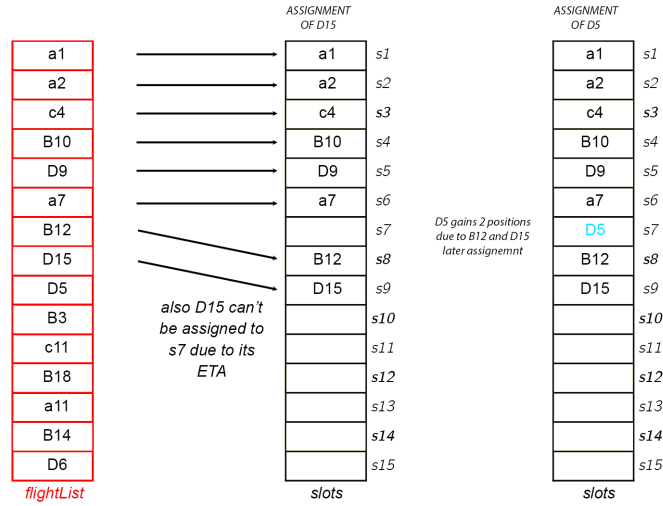


Figure 8: Assignment of flight D15

*As from this stage, there are no other ETA conflict, all other flights are smoothly assigned to the first free slot.*

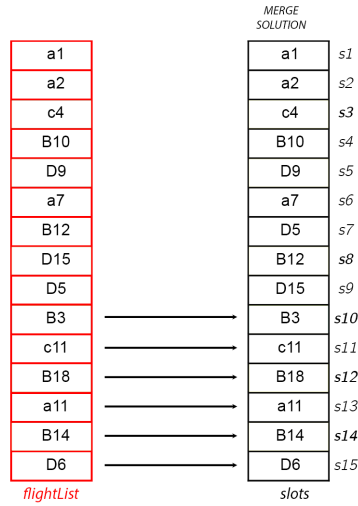


Figure 9: Last assignments

## 5 Doubts

This work aimed to provide a detailed analysis of the UDPP mechanism described in the document **SESAR Solution PJ07.02 SPR-INTEROP/OSD**

for **V2 - Part I** and an high-level programming language description of the UDPP algorithms. During the analysis some doubts arose and therefore some parts of the algorithms have been reinterpreted to overcome the issues and hopefully achieve the same goals. Below are explained the two main dilemmas which came across, with the relative document references.

## 5.1 Bflights

The idea should be that *Bflights* are essentially exempted from the reallocation and that the corresponding slots are consequently not available for the other flights reallocation. This seems to arise already in the definition of section **2.4.6.3.3 UDPP FDR feature** (“*B* Baseline priority: specifies to keep the baseline delay of the flight as the current target delay.”) and remarked in section **2.2.6.3.8 AU inputs: different ways of thinking** (“N.B.: *B* can be specifically set to a flight if the AU wants to exclude the flight from the reordering: keep Baseline as a solution.”). The exclusion of *Bflights* from the slot list explicitly appears in *Rule.8* of section **3.3.1.1.1 Step 1 - Manage Protected flights (Pflight) (with Margins or not)** and in **3.3.1.1.2 Step 2 - Manage Margin Flights** (*The candidate Slot for the Margin flights can be found on all the available slot of the AU except the one already assigned to the Pflights and the one with an explicit B value (explicit Baseline priority value given on a flight).*). However, they are not mentioned in the definition of the *Manage Nflights* function of section **3.3.1.4 Step 4 - Manage the priority value flights (Nflights)** (“Nflights are defined by the fact that they have no Margins defined on its and not “P” and not “S” as priority value. Lflights are part of Nflights management but with a priority = to 1000”); in this case they should also not be included as they have no priority assigned which would make impossible to apply any FDR procedure on them.



## 5.2 Function Manage Time Solution



In this work the role of this function is played by the function *AssignOrShiftEarlier* (Algorithm 4) which should provide the same result. A direct implementation of the *Function Manage Time Solution* function wasn’t possible as there were the following two doubts concerning its sub-functions:

### 5.2.1 Manage Solution Earlier

In section **3.3.1.1.7 Function - Manage Solution Earlier** it is not clear to us what the cycle is looping over and if the call to *Move flight Earlier* is made at each iteration or at the end of the loop.

### 5.2.2 Move Flight Earlier

Despite the aim of this function is comprehensible, according to how is defined, we are not completely sure about what is actually doing.

## References

- [1] N. Pilon, A. Cook, S. Ruiz, A. Bujor, and L. Castelli, “Improved flexibility and equity for airspace users during demand-capacity imbalance-an introduction to the user-driven prioritisation process,” *Sixth SESAR Innovation Days*, 2016.