AN AIR TRAFFIC SIMULATION MODEL THAT PREDICTS AND PREVENTS EXCESS DEMAND

Dr. Justin R. Boesel*
The MITRE Corporation
Center for Advanced Aviation System Development (CAASD)
McLean, Virginia 22102

ABSTRACT

An airplane's ability to absorb delay while airborne is limited and costly. Because of this, the air traffic management system anticipates and manages excessive demand for scarce shared resources, such as arrival runways or busy airspace, so that the delay necessary for buffering can be spread out over a larger distance, or taken on the ground before departure. It is difficult to model these important dynamics in a standard queue-resource simulation framework, which does not account for limited delay absorption capacity. The modeling methodology presented here captures these dynamics by employing a large number of independent threads of execution to monitor and enforce a large number of relatively simple mathematical relationships. These relationships calculate feasible time windows for each portion of each flight. The model was implemented in the SLX simulation language. The speed and scalability of SLX are essential to the approach, which would otherwise be impractical.

INTRODUCTION

From a capacity modeling perspective, airplane traffic is fundamentally different from automobile, rail or ship traffic; while a car, train, or ship can stop and wait for an essentially unlimited amount of time in the middle of its journey, an airplane cannot. In other words, once airborne, an airplane's ability to absorb delay is limited and costly, and it is important in modeling delays and capacities.

Because of this limited delay absorption capacity, the air traffic management system anticipates and manages excessive demand for scarce shared resources (e.g., arrival runways, busy terminal airspace) so that the delay necessary for buffering can be spread out over a larger distance, or taken on the ground before departure. These actions, however, can ripple back and block resources upstream, such as departure runways and busy sectors.

It is difficult to model these dynamics in a standard queue-resource simulation framework. In a standard queue-resource model, there is no concept of limited delay absorption capacity. For instance, in a factory setting, a part moving from one work station to another may wait for one minute or one week before receiving service. The *number* of parts waiting for service (queue size) may be explicitly limited, but the wait time per part is not.

To model airplane traffic, one needs to be able to anticipate excessive demand for a resource well *before* it occurs, so that the flight to be delayed has adequate distance over which to absorb the required delay.

This paper describes two different, but closely related, mechanisms for anticipating and preventing excessive demand in a simulation model. One mechanism is a link-node network that treats each node as a resource that can handle one flight at a time. The other model, which essentially sits on top of the link-node model, is a sector model that considers a sector as a resource that has different capacities corresponding to different resources (e.g., communications and coordination) and can handle many flights simultaneously. In the sector model, each flight can place a different burden on each of the sector's capacities.

The remainder of this paper is organized as follows: the next section describes how aircraft are delayed for buffering; the third section explains why it is important to capture the dynamics caused by limited buffering capacity; the fourth section provides an overview of the link-node network model; the fifth section describes the central "monitor and enforce" mechanism used in the simulation; the sixth section provides an example modeling two merging aircraft; the seventh section provides an overview of the sector capacity model; the eighth section describes the model in more detail; the ninth section describes elements of the Simulation Language with Extensibility (SLX) simulation language that are central to this approach; and the tenth section summarizes and draws some conclusions.

DELAY ABSORPTION BUFFERING

In almost any capacity-constrained system, the ability to buffer demand during busy periods is key to increasing utilization of scarce server resources. For

* Sr. Simulation and Modeling Engineer

instance, during lunchtime at a fast food restaurant, customers wait in line while the cashier/server takes orders from other customers. When one customer is done, the next in line receives service, and the server remains busy. As wait time increases, a customer can choose to remain in line or can opt out, and leave the restaurant. In the air traffic control system, some similar situations exist: departing airplanes wait on taxiways for their turn on a runway. Once aircraft are in the air, however, the situation changes. An aircraft cannot opt out of landing and, because of fuel constraints, it cannot wait for an arrival runway indefinitely. Within these constraints, buffering airborne flights—making them wait in the air—even for relatively short periods of time is costly in a number of ways.

- Buffering requires controller work. Overloading a controller is undesirable because it can compromise safety.
- Buffering usually increases mileage, which burns fuel and increases aircraft wear and tear.
- Buffering requires airspace.

Air traffic managers have four basic methods in which they can delay aircraft to keep them from overwhelming a resource such as a runway or a controller downstream. These methods, and their relative costs and benefits, are described as follows:

1. *Ground Delay*
   Delaying a flight on the ground before it departs is relatively cheap in terms of fuel and controller workload, even though it is not free to the airlines, their passengers or cargo. Ground delay can absorb practically unlimited amounts of time. Ground delay is often used to prevent congestion in the air, which, if left unchecked, could overburden controllers and compromise safety. Because of departure runway congestion and flight-time variability, however, it is not practical to use ground delay to fine-tune a flight's arrival time at a distant airport.
2. *Airborne Holding*
   Placing a flight into airborne holding is expensive in terms of fuel and controller workload. Furthermore, holding requires reserved airspace; the locations at which airborne holding can take place are limited. Holding is most commonly used to delay arriving flights close to (within 50 miles of) their destination airport. Despite the costs, holding allows controllers to delay airborne flights for relatively large amounts of time (tens of minutes). At large airports with relatively unconstrained airspace, such as Atlanta Hartsfield, controllers use the buffering capacity provided by airborne holding to make more efficient use of arrival runways.[1]
3. *Vectoring*
   Vectoring means extending a flight's path (thereby delaying it) by turning it. Vectoring allows controllers to delay aircraft more precisely and with less expense, in terms of fuel cost, than they could with airborne holding. The amount of delay that can be achieved with vectoring is closely related to the amount of airspace a controller can use. Vectoring is a very common technique, especially for sequencing flights onto an arrival runway.
4. *Speed Control*
   Slowing a flight down to delay it requires little airspace, but the amount of delay that can be absorbed with speed control is not great.

WHY MODEL LIMITED DELAY ABSORPTION?

Because airborne flights can absorb only a limited amount of delay, buffering caused by contention for a downstream resource, such as an arrival runway, can quickly ripple back upstream and cause congestion in an upstream resource, such as an en route sector. The subsequent congestion upstream can delay departures from and arrivals to other airports.

If a model fails to capture the limits on delay absorption, it will miss these blocking effects upstream. This makes it important to model aircraft taking delay not only in the correct amount, but also at the correct place and time.

Figure 1 illustrates this problem. Suppose airport D sends flights to airport A, and airport C send flights to Airport B, all via en-route Sector Y. If runway congestion at Airport A delays arrivals, they may spend more time in Sector Y, especially if the delay absorption capacity between Y and A is small. If this causes Sector Y to become too busy, departures from C may be held on the ground. A model that accounts for limited delay absorption would capture this dynamic.

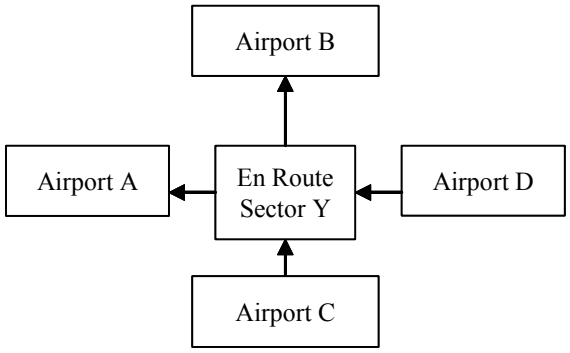

Figure 1: En Route Sector as a Constraint

On the other hand, in a model that overstates the delay absorption capacity of flights between Sector Y and Airport A, the arrivals to A will quickly pass through Sector Y, which will not become too busy, thus allowing departures from C to B to proceed undelayed. This model will understate delay.

OVERVIEW OF THE LINK-NODE MODEL

In the first model presented here, aircraft move along a link-node network structure. Each flight requires a minimum time to traverse each link, and each flight can absorb only a limited amount of additional delay on each link. To represent an airspace that has plenty of room for vectoring, this maximum delay parameter can be set high, while a narrower, more constrained airspace would have a lower maximum delay parameter.

As flights merge onto common links or cross each other's paths, minimum separation between aircraft is maintained. Each link and node in the model is assigned a minimum required separation (in minutes) that defines its capacity.

The model is essentially a network model with nodes, links, and flights moving and absorbing delay on links. Unlike most network models, however, this model anticipates contention for resources long before it occurs and spreads the required delay absorption out across the links, rather than in a buffer immediately in front of the constrained resource. Four object types define the model:

1. *Flight Object*
   Each Flight object represents a single flight. It has the flight's aircraft ID, aircraft type, desired departure time, and a flight plan, defined as a list of Links.
2. *Link Object*
   Flights use Links to get from one place to another. A Link object can be used to represent an airway at a particular altitude. Links are defined to be one-way only, and while a Link can be shared by several Flights, passing is not permitted on a Link. In other words, Links impose a strict First-In-First-Out (FIFO) policy on Flights. Each Link has pointers to its starting and ending Nodes, pointers to all of the Flights that will pass over it, and minimum required separations (in minutes) that define its capacity.
3. *Node Object*
   A Node object is used to connect Links. A Node object, which can be thought of as a point in 3-D space, can be used to represent a waypoint or a fix at a particular altitude. Nodes represent crossings, merges, and split relationships between Link objects. Each Node has a list of the Links coming into and out of it. Like a Link, a Node has several minimum required separations that define its capacity.
4. *Flight-By-Link Object*
   As each Flight crosses each Link on its flight path, the model generates a great deal of timing information. Flight-by-Link objects keep track of all of this information. A Flight-by-Link object represents a particular Flight on a particular Link. For example, if a Flight has *n* Links on its path, then *n* Flight-By-Link objects will be created for that Flight.

The Flight-By-Link object is the workhorse of the simulation. It has pointers to three other Flight-by-Link objects, which define the object's relationship with the rest of the model. One pointer refers to the Flight immediately ahead of it on the same Link, and the other two pointers refer to the same Flight on the next and previous Links. Figures 2 and 3 illustrate these relationships.

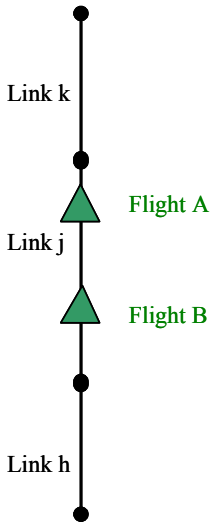

Figure 2. Flights A and B on Links h, j, and k

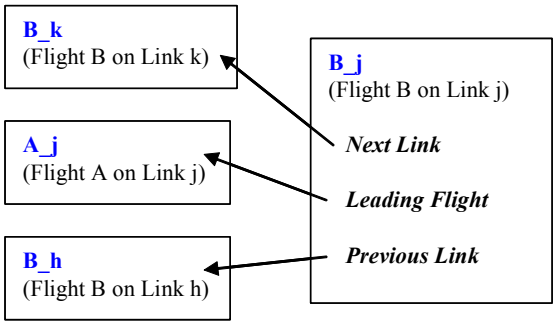

Figure 3. Flight-By-Link Object B_j's Pointers to Adjacent Flight-By-Link Objects

Suppose Flight B follows Flight A across Links h, j, and k, as shown in Figure 2. To represent this in the model, one would need Flight objects for A and B, and Link objects for h, j, and k. To represent the flights' movement over these Links, one would need to create six Flight-By-Link objects, A_h, A_j, A_k, B_h, B_j, and B_k. Figure 3 illustrates the pointer relationships of Flight-By-Link B_j to its "adjacent" Flight-By-Link objects B_k (same Flight, next Link), A_j (leading Flight, same Link), and B_h (same Flight, previous Link).

Each Flight-by-Link object has two quantities—minimum traversal time and maximum delay—that determine the minimum and maximum amounts of time the Flight can spend on the Link. The minimum traversal time represents the amount of time a Flight needs to cross a Link, and the maximum delay represents the amount of additional time a Flight could absorb on a particular Link.

MONITOR AND ENFORCE

In the model presented here, each object monitors and enforces a number of mathematical relationships. The following example illustrates such a relationship, and describes its enforcement.

Suppose the two Flights depicted in Figure 2, need to increase separation before entering Link k. Let T_A and T_B be the clock times at which Flights A and B (respectively) enter Link k, and let S_k be the minimum required separation (in minutes) between Flights entering Link k. To respect this separation, one must ensure that:

$$T_B \geq T_A + S_k \qquad (1)$$

Flight-By-Link object B_k can read all three quantities in (1): T_B is local to B_k, and the others can be read via pointers. B_k gets its own independent thread of execution to enforce the separation requirement expressed in (1), illustrated by the following pseudo-code:

```
fork {// fork command creates new thread

    forever //Enters Loop
{
    wait until (TB <  TA + Sk);
    //relationship is violated

    TB = TA + Sk;
    //Enforce relationship

    }//end forever loop
}//end fork
```

The new thread immediately enters a "forever" loop, the only purpose of which is to wait until inequality (1) is violated, and then to correct it. This basic mechanism monitors and enforces a mathematical relationship.

At its heart, the model is essentially a system of hundreds of thousands of such relationships, each one monitored and enforced by its own independent thread and a "wait until" statement.

MODELING MOVEMENT AND LIMITED DELAY ABSORPTION: AN EXAMPLE

The following example illustrates how the objects and the monitor-and-enforce mechanism work together to model delay pass back due to limited delay absorption.

Consider Figure 4. Suppose Flight C traverses the airspace represented by Links v, w, and z, and Flight D, which starts just a little bit later, traverses the airspace represented by Links x, y, and z. Suppose that the Flights will need to be separated by one minute when crossing their merge point, S.

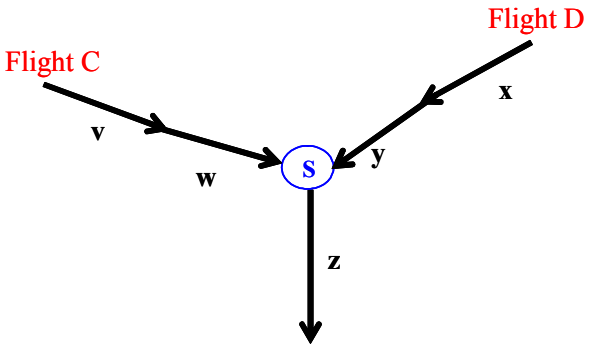

Figure 4: Merging Flights at Point S

Suppose further that each Flight can traverse each Link in a minimum of two minutes, but can only absorb 45 seconds (0.75 minutes) of delay on each Link. To respect the separation requirement at point S, one of the Flights will need to be delayed.

To model this, one creates Flight objects representing C and D, Link objects representing v, w, x, y, and z, a Node object representing point S, Flight-By-Link objects C_v, C_w, and C_z, representing Flight C on Links v, w, and z, and Flight-By-Link objects D_x, D_y, and D_z representing Flight D on Links x, y, and z.

Each Flight-By-Link object has three pointers to other Flight-By-Link objects that are "adjacent" (in space or sequence):

- The *Next Link* pointer points to the same flight on the next Link in the flight plan. For instance, C_v has a pointer to *Next Link* C_w

- The *Previous Link* pointer points to the same flight on the previous Link in the flight plan. For instance, C_w has a pointer to *Previous Link* C_v
- The *Leading Flight* pointer points to the previous flight on the same Link. In this example, there is only one such meaningful instance of this; D_z has a pointer to *Leading Flight* C_z

Each Flight-By-Link object has several attributes, specifically:

- *Minimum Traversal Time* – two minutes for all Flight-By-Link objects in this example
- *Maximum Delay* – 45 seconds for all Flight-By-Link objects in this example
- *Best Guess Time of Entry* – the current best estimate of when (in simulation clock time) the Flight will enter the Link;
- *Best Guess Time of Exit* – the current best estimate of when (in simulation clock time) the Flight will exit the Link

To model simple movement across a single Link, there is a thread that monitors and enforces the relationship (M1):

Best Guess Time of Exit ≥
Best Guess Time of Entry +
Minimum Traversal Time

To model movement from Link to Link, there is a thread that monitors and enforces the relationship (M2):

Best Guess Time of Entry ≥
***Previous Link*'s** *Best Guess Time of Exit*

Each Link object has a constant *Minimum Separation Required at Entry*, which is used to separate merging and in-trail Flights. This is set to one minute on Link z. Each Flight-By-Link object compares its *Best Guess Time of Entry* to that of the Flight in front of it. The mechanism for doing this is a thread that monitors and enforces the relationship (M3):

Best Guess Time of Entry ≥
***Leading Flight*'s** *Best Guess Time of Entry* +
Shared_Link's
Minimum Separation Required at Entry

To pass back constraint information from Link to Link, there is a thread that monitors and enforces the relationship (M4):

Best Guess Time of Exit ≥
***Next Link*'s** *Best Guess Time of Entry*

To model the limited delay absorption capacity on each Link, there is a thread that monitors and enforces the relationship (M5):

Best Guess Time of Entry ≥
Best Guess Time of Exit –
(*Minimum Traversal Time* +
Maximum Delay)

Suppose Flight C starts at time 100.0 and Flight D starts at time 100.1. The threads enforcing relationships M1 and M2 will cascade forward through the system, setting *Best Guess Time of Entry* at 104 for Flight-By-Link object C_z (see Figure 5).

A similar chain of events takes place on objects D_x, D_y, and D_z, so that D_z's *Best Guess Time of Entry* will equal 104.1. But because the entry times are less than one minute apart, this causes a separation violation on entry to Link z. This violation is caught by M3, which will force D_z's *Best Guess Time of Entry* to 105, causing M4 to force D_y's *Best Guess Time of Exit* to 105 (see Figure 6).

Flight D needs to absorb a total delay of 0.9 minutes overall, but only 0.75 minutes can be absorbed on any single Link. M5 forces D_y's *Best Guess Time of Entry* down to 102.25. This means that 0.75 minutes of delay, the maximum, is absorbed on Link y. The remaining 0.15 minutes is pushed back to Link x by M4 which forces D_x's *Best Guess Time of Exit* to 102.25.

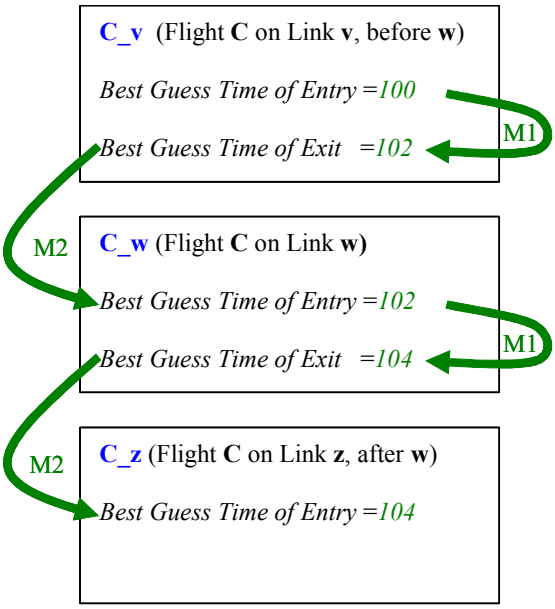

Figure 5: Sending Flight C's Timing Information Forward via Flight-By-Link Objects C_v, C_w, and C_z

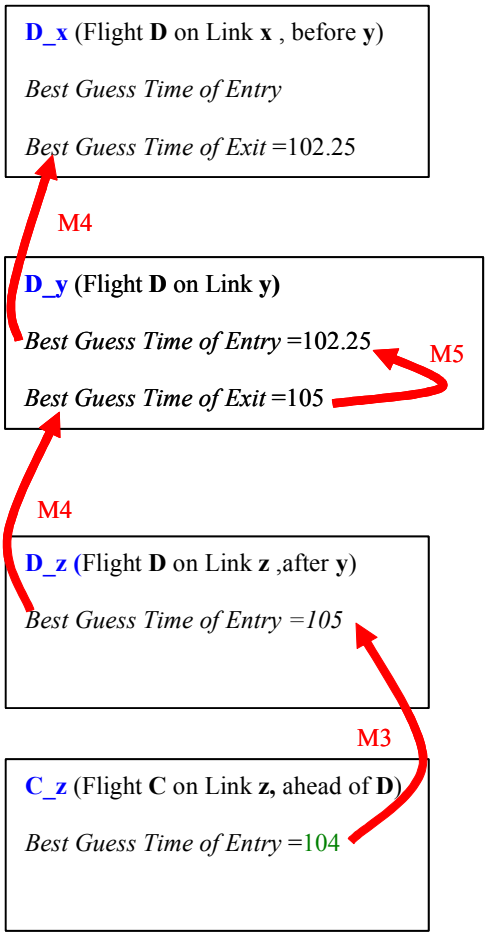

Figure 6. Flight C Delays Flight D, and Delay is passed back via Flight-By-Link objects D_z, D_y, and D_x.

At this point, the monitoring and enforcing stops, because D_x's *Best Guess Time of Exit* of 102.25 is well within bounds: M5 is unviolated because D_x's *Best Guess Time of Entry* is 100.1, its *Minimum Traversal Time* is two, and its *Maximum Delay* is 45 seconds (0.75 minutes).

Note that these changes, which essentially put the system of equations back into balance, take place in zero simulated time. In this case, the changes should occur *before* the simulation clock reaches 100, the time at which the Flights start on their first Links. The simulation clock advances to the next scheduled "Link movement", that is, the next scheduled *Best Guess Time of Entry* among all of the Flight-By-Link objects defined in the model. Subsequent balancing actions occur between every advance of the simulation clock.

Because each Flight-By-Link object only has to directly account for the three "adjacent" Flight-By-Link objects, the model scales up well. Once one has set up the rules governing these relationships, changes that take place far away are allowed to ripple through the system automatically.

Crossing Flights

The previous example illustrates the fundamental monitor-and-enforce mechanisms and the system of relationships required to simulate a merge. Flights whose paths cross, rather than merge, however, share only a common Node object, not a common Link object. As a result, an additional set of relationships among Flight-By-Link objects is required. Specifically, whereas a merging Flight requires a pointer to the leading Flight on the shared Link, a crossing Flight requires a pointer to the leading Flight through the shared Node. Apart from that, the mechanisms for maintaining separation and absorbing delay due to crossing are practically identical to those for merging.

OVERVIEW OF SECTOR CAPACITY MODEL

The Link-Node network model can represent the important problem of flight separation and delay pass-back with limited delay absorption. There are, however, other important constraints of the air traffic control system that the link-node network model cannot capture. Specifically, the limited capacity of a sector's control team is often more constraining than any flight separation requirement. For instance, a sector may be unable to handle an additional flight because the control team does not have time to communicate with the flight, even though there may be plenty of air space available to separate the flights.

A sector's control team may also face constraints on its ability to coordinate with other sectors, or on the number of flights the team can track simultaneously. Furthermore, the demands placed on each of these capacities may differ from one flight to the next. For instance, it may be quite easy to keep track of a routine flight, while an "oddball" flight may require a great deal more controller attention.

In the model presented below, each sector is a resource that can handle many flights simultaneously, and each flight traverses several sectors, placing various demands on each one it crosses. Furthermore, each sector has several finite capacities, such as communications and coordination, rather than just a single one, such as flight count. These capacity and demand numbers are user *inputs* to the model, and the model imposes delays upon flights to keep each sector within its capacity. The model makes no attempt to account for individual controller instructions or pilot actions.

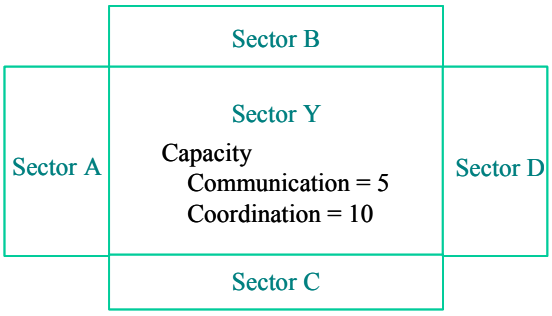

Figure 7. Sectors A, B, C, D and Y

For example, let us suppose that in Figure 7, Sector Y is assigned a communications capacity of five, and a coordination capacity of ten. Furthermore, suppose each flight going from Sector C to Sector B (a C-to-B flight) places a demand of two communications units and three coordination units on Sector Y, while each flight going from Sector D to Sector A (a D-to-A flight) places a demand of one communications unit and four coordination units on Sector Y. Sector Y can simultaneously handle any combination of C-to-B and D-to-A flights that does not, in aggregate, violate any of Sector Y's capacities. For instance, Sector Y could simultaneously handle two C-to-B flights and one D-to-A flight, which would place a total demand of five (that is, 2x2+1x1) communications units and ten (that is, 2x3+1x4) coordination units. The sector could not, however, handle one C-to-B flight and two D-to-A flights simultaneously, which would place a total demand of 4 (that is, 1x2+2x1) communications units and 11 (that is, 1x3+2x4) coordination units, exceeding the sector's coordination capacity. In this case, one of the flights would need to be delayed until another flight exited the sector, keeping the demand within the capacity.

Just like the Link-Node model, the Sector model prevents overloads by delaying flights, and it must impose these delays early enough so that the Flight has enough distance over which to absorb the delay. The mechanisms used to predict and react to sector overload are, however, quite different from those used to maintain separation in the Link-Node model. The Link-Node model rests heavily on two assumptions: one is the assumption that a Node can handle only one Flight at a time, and the other is that Links impose a FIFO policy on Flights. Roughly speaking, under these assumptions, if each Link-By-Flight object knows where it is going and who is ahead of it, and if each Node knows who wants to go through it, the model has all the information it needs to maintain separation.

A sector, however, cannot assume a FIFO policy and must be able to handle multiple flights simultaneously. To manage demand under these conditions, the Sector model monitors demand at fixed future time horizons (e.g., ten minutes ahead of the simulation clock), and imposes delay in reaction to predicted demand overloads.

Despite these differences, the Sector model is tightly integrated with the Link-Node network model. The Sector model sits on top of the Link-Node model, using the Link-Node network infrastructure to:

- move flights
- predict each flight's sector entry and exit times
- propagate delay backward through the system

ELEMENTS OF THE SECTOR MODEL

Implementing the Sector model requires several additional object types. Three objects are most important: the Sector object, which defines the sector and its capacities, and responds to predicted overloads; the Flight-By-Sector object, which represents the timing and demands of a single Flight on a single Sector, and connects the Sector model to the Link-Node model, and; the Horizon Monitor object, which tells the Sector object whether a particular Flight will be in the Sector at a particular time in the future.

Sector Object

A Sector object contains a set of Link objects and a vector of resource capacities (e.g., five units of communications, ten units of coordination). The set of Links represents the sector control team's area of work. Any Flight that is on a Sector's Link will place a demand on the Sector. A Link cannot be a member of more than one Sector object.

In Figure 8, Sectors A, B, C, and D each have one Link (a, b, c, and d, respectively) while Sector Y has four links (y1, y2, y3, and y4).

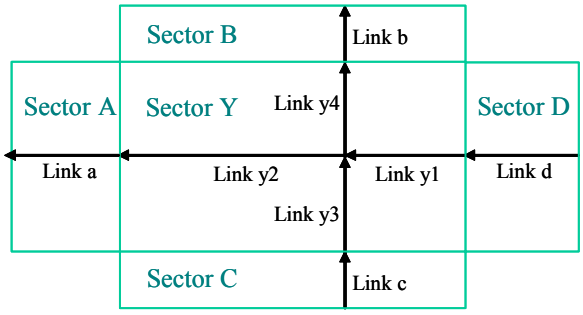

Figure 8. Sectors A,B,C,D and Y, and their Links

The Sector Object maintains sets of expected (future), current, and past flights. These are updated as the simulation progresses. The Sector Object also contains a matrix of projected demands for each resource at each future time horizon. Table 1 shows an example of this

for a Sector with two capacities (communications and coordination) and two lookahead horizons (10 minutes and 20 minutes). The values in this table are incremented and decremented by Horizon Monitor objects, described later.

Table 1. Example of Predicted Demand on a Single Sector With Two Lookahead Horizons

	Horizon (minutes into the future)	
Capacity Measure	10 min.	20 min.
Communications	4 units	6 units
Coordination	5 units	7 units

Each cell in the table (each resource at each horizon) is monitored by a thread that compares projected demand to capacity. If demand exceeds capacity, the model delays the last flight that is within the horizon by increasing a Flight-By-Sector variable, called Earliest Allowed Time of Entry, by a small amount. This action is repeated until the last flight is pushed outside of the horizon. If demand still exceeds capacity, the (new) last flight within the horizon is pushed outside of the horizon. This action is repeated until predicted demand is within capacity. To determine which flight is the last within the horizon (that is, the one that will take the delay), the model runs through the Sector object's set of expected flights before imposing each delay. This means that after one flight gets pushed out of the horizon, delay will be imposed on the next one.

Flight-By-Sector Object

Just as the Link-Node model has a Flight-By-Link object to represent a particular Flight on a particular Link, the Sector model has a Flight-By-Sector object to represent a particular Flight in a particular Sector. The Flight-By-Sector object has pointers to the Flight and to the Sector, and contains the flight's demands for each Sector resource (e.g., this Flight demands one unit of communications and three units of coordination from this Sector).

The Flight-By-Sector object connects the Sector model to the Link-Node model. Each object maintains a Flight's predicted Sector entry and exit times by "piggy-backing" on Flight-By-Link Objects. To do this, the Flight-By-Sector object maintains pointers to the Flight-By-Link corresponding to the flight's first and last Links in the Sector. These could both be the same Link, such as Link a in Sector A in Figure 8, or different Links, such as Links y1 and y2 in Sector Y.

In Figure 8, suppose we have a single flight that traverses three Sectors (D, Y, and A) on four Links (d, y1, y2, and a). The model would create a Flight-By-Link object for each Link, and a Flight-By-Sector Object for each Sector. The flight's predicted sector entry time for Sector Y is the same as its predicted entry time on to Link y1, and its predicted sector exit time is equal to the predicted exit time from Link y2.

To maintain the Flight's predicted Sector entry time, the Flight-By-Sector object has one thread that monitors and enforces the relationship:

Sector Entry Time ==
Best Guess Time of Entry on First Link

and another thread that monitors and enforces the relationship:

Sector Exit Time ==
Best Guess Time of Exit on Last Link

In addition to reading times *from* the Link-Node network, Flight-By-Sector objects also transmit Sector-based delay *to* the Link-Node network. Each Flight-By-Sector object has a variable called "Earliest Allowed Time of Entry". When a Sector wants to delay a Flight, it increases this variable. To communicate this delay to the Link-Node Network, the Flight's first Flight-By-Link object in the Sector (e.g., Link y_1 in Figure 8) has a thread that monitors and enforces the following relationship:

Flight-By-Link's Best Guess Time of Entry $\geq$
Flight-By-Sector's Earliest Allowed Time of Entry

The Sector imposes the delay on the Link, and the Link-Node network takes care of the rest, propagating delay backward if necessary. Sector objects do not communicate directly with one another, but they communicate with their Links, and one Sector's Links can communicate with another Sector's Links.

Horizon Monitor Object

The model employs several "lookahead horizons" (e.g., ten minutes into the future), future points in time at which the model monitors the predicted demand upon the sector. Figure 9 illustrates the concept. Suppose Flights Q, R, and S are approaching a Sector. The brackets represent the time each flight will be in the sector. If current simulation clock time = t, then Flight Q has just entered the Sector, Flight R will enter the Sector before time = t + horizon, and Flight S will not enter until later.

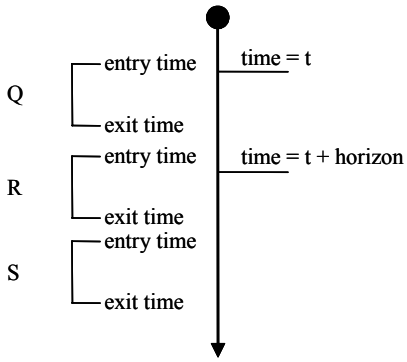

Figure 9. Flights Q, R, and S Sector Entry and Exit Times Spread Over the Time Line

Suppose that each Flight places a demand of two communication units on the Sector, which has a communications capacity of five. In Figure 9, there is no problem, because there will not be more than one flight in the sector at any given time. Now consider Figure 10. As the dotted line indicates, Flights T, U, and V will all be in the Sector at time = t+ horizon. This would overload the Sector's communication capacity, so a Flight will need to be delayed. In Figure 11, we see that Flight V's Sector entry time has been delayed past the horizon, but there will still be a sector overload after that time. In Figure 12, the simulation clock has advanced to time = t+s, and Flight V's entry time has been delayed until time = t+s+horizon, which is Flight U's Sector exit time. Now, the demand is spread out over time, and will remain within the Sector's capacity.

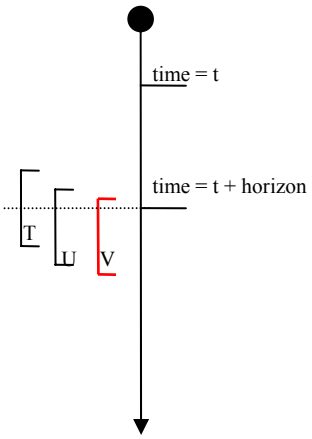

Figure 10. Flight T, U, and V all in the Sector at time = t + horizon

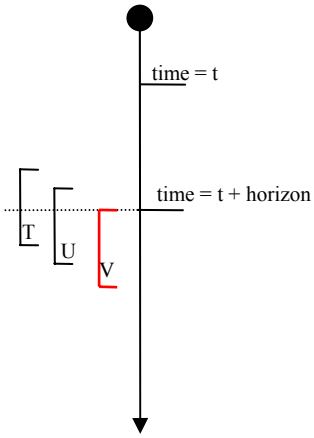

Figure 11. Flight V's Sector Entry Time Delayed to time = t+ horizon

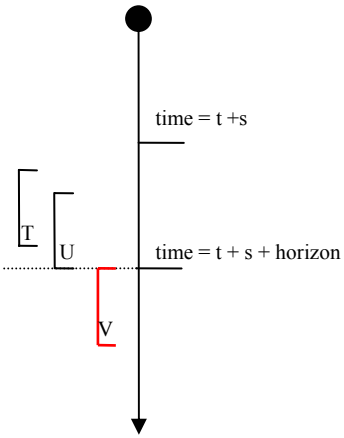

Figure 12. Simulation Clock advances to time = t + s and Flight V's Sector Entry Time Delayed to time = t + s + horizon

To keep track of when a Flight's predicted time in a Sector rolls into (and out of) a particular time horizon, we create a "Horizon Monitor" object for each Flight-By-Sector object and each lookahead horizon. So if there are ten flights passing through ten sectors each with three lookahead horizons, the model will, over the course of a simulation, create 300 Horizon Monitor objects. The Horizon Monitor object also increments and decrements predicted Sector demand (that is, the cells in Table 1), each time a Flight-By-Sector object rolls into or out of a particular horizon.

A Flight-By-Sector's "time window" (the time it plans to spend in the Sector) can be grouped into one of three states with regard to a particular horizon. The window can be *before* the horizon (that is, predicted entry time is *later* than t + horizon), *passed* the horizon (that is, predicted exit time is *earlier* than t + horizon), or *on* the horizon (entry time is *earlier* than t + horizon, and exit time is *later* than t + horizon).

In the model, a Flight-By-Sector's time window is pulled forward (with regard to a particular horizon) as the simulation clock advances, but the time window is also pushed backward as the model imposes delay upon Flights, increasing predicted sector entry and exit time. To keep track of this, the Horizon Monitor employs four threads.

The first thread is designed to detect the Flight entering the horizon as simulation clock (t) advances. It waits until:

Predicted Sector entry time < t + horizon
AND
Time Window's State is before horizon

When these conditions are met, the thread *adds* the Flight's demands to the Sector's totals and switches the time window's state to *on* the horizon.

The second thread is designed to detect the Flight exiting the horizon as simulation clock (t) advances. It waits until:

Predicted Sector exit time < t + horizon
AND
Time Window's State is on horizon

When these conditions are met, the thread *subtracts* the Flight's demands from the Sector's totals and switches the time window's state to *passed* the horizon.

The third thread is designed to detect the Flight being pushed back into the horizon as delay is imposed on the Flight. It waits until:

Predicted Sector exit time > t + horizon
AND
Time Window's State is passed horizon

When these conditions are met, the thread *adds* the Flight's demands from the Sector's totals and switches the time window's state to *on* the horizon.

The fourth thread is designed to detect the Flight being pushed back out of the horizon as delay is imposed on the flight. It waits until:

Predicted Sector entry time > t + horizon
AND
Time Window's State is on horizon

When these conditions are met, the thread *subtracts* the Flight's demands from the Sector's totals and switches the time window's state to *before* the horizon.

FUNDAMENTALS OF THE SLX SIMULATION LANGUAGE

A model employing the concepts described previously was implemented in the SLX simulation language. SLX (Wolverine Software) is a PC-based simulation language and development environment that has some unique capabilities that enable the user to build and run very large, complex models.[2] While a detailed description of the language is beyond the scope of this paper, three fundamental aspects of the language, necessary for implementing the delay absorption model, are described as follows.

- *Pucks*
  An SLX puck is like an independent thread, or stream of execution, within the simulation model. Pucks allow the modeler to simulate many things happening in parallel. The SLX pucks run extremely quickly, and each one consumes relatively little memory. Each of the mathematical relationships in the model presented previously was monitored and enforced by its own puck. A model employing the concepts presented in this article used over 500,000 pucks, up to 250,000 running simultaneously, and ran in about 9 seconds on a PC with a 2-gigahertz processor.

- *Wait Until*
  SLX has a "wait until" statement that allows the user to model time- and state-based delays. While a state-based delay capability is not unique to SLX, the speed with which it executes is noteworthy. Almost all of the 500,000 pucks in the model mentioned previously were controlled with "wait until" statements.

- *Active Objects*
  SLX is an "object-based" language that allows each object to have an unlimited number of independent pucks (threads of execution), thereby making the object "active." This means that the user can model several things happening in parallel within a single object. Each Flight-By-Link object in the model described previously had multiple pucks, one puck for each of the mathematical relationships monitored and enforced.

These modeling constructs, combined with the speed and scalability of SLX, open up many modeling possibilities—like the approaches described previously—that would otherwise be impractical.

CONCLUSIONS

This article described a methodology for modeling limited delay absorption capacity, which is a fundamental problem in air traffic management. At its heart, the methodology is based on a large number of independent streams of execution monitoring and enforcing a large number of relatively simple mathematical relationships. This approach relies heavily on the speed and scalability of the SLX simulation language.

ACKNOWLEDGMENTS

The author would like to acknowledge the important contributions of Michael White of The MITRE Corporation. Mr. White devised the original conceptual model behind the multi-capacity sector.

REFERENCES

1. Voss, William R, and J. Hoffman. 2001. Analytical Identification of Airport and Airspace Capacity Constraints. In *Air Transportation Systems Engineering*, ed. G.L. Donohue, A.G. Zellwegger, H. Rediess, and C. Pusch, 409-419. Reston, Virginia: American Institute of Aeronautics and Astronautics.
2. Henriksen, James O. 1998. Stretching the boundaries of simulation software. In *Proceedings of the 1998 Winter Simulation Conference*, ed. Medeiros, D.J., E. Watson, M.S. Manivannan, and J. Carson, 227-234. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.