Curtin University — Department of Computing
Sri Lanka Institute of Information Technology

Distributed Computing (COMP3008)

Semester 1, 2017

Assignment

Due: Friday 9 June at 5:00 pm

Weight: 20% of the unit mark.

1 Overview

Your task is to write a distributed program that simulates Air Traffic Control (ATC) for some of Australia's major domestic airports (Perth, Melbourne, Sydney and Brisbane). It must track a set of planes flying along specific routes between these airports.

In reality, each airport would have its own system for controlling arriving and departing airplanes. Our simulation will approximate this, by having a separate ATC server representing each airport. They will be coordinated by a central "master" server.

The simulation will work on 15-minute simulated time intervals, coordinated by a "master" server (15 *simulated* minutes, that is, not real minutes). During each interval, the "master" contacts the ATC ("slave") servers to request that they update the simulation.

These updates include tracking the position and fuel of all airplanes, deciding which airplane should land (if any), which airplane should take off (if any), and which airplanes need to be "handed over" to a different ATC server. (Hand-over between ATC servers occurs when a plane departing one airport approaches within 300km of its destination airport.)

There is a lot of detail to this, so read the following pages carefully!

2 Simulation Data and Logic

2.1 Airplane State

Each airplane in the simulation is always in one of the following five states:

Landed – The airplane is on the ground at an airport.

In-Transit – The airplane is in-flight to the destination airport, but is still 300km or more away.

Entering Circling – The airplane is within 300km of the destination airport, but has not yet arrived.

Circling – The airplane is at the destination (0km), but is flying in a holding pattern waiting for other airplanes to land.

Crashed – The airplane ran out of fuel and the pilots bailed (not enough parachutes for the passengers unfortunately).

All airplanes start the simulation on the ground ("Landed") at a particular airport.

While on the ground, airplanes use no fuel, and are refueled according to how far they will need to travel to the airport they are next scheduled to go to. An airplane spends at least one hour (four 15 minute timesteps) on the ground before it is ready for takeoff. It may spend longer if it needs to wait for other airplanes to take off first.

2.2 Queueing to Land

At any given time, several planes may have arrived a particular airport. However, only one (at most) can land during each 15 minute interval (although it may do so while another airplane is taking off), and the rest will have to circle. Landing uses up the 15 minute interval for that airplane.

The airplane with the lowest fuel must land first.

An airplane *may* arrive and immediately land in the same time interval. That is, it can transition directly from entering-circling to landed, without ever going through the circling state (provided it has less fuel than any other airplanes that could also land).

2.3 Refuelling and Taking Off

Only one airplane (at most) may take off from each airport during each 15 minute interval (although it may do so while another airplane is landing). Taking off uses up the 15 minute interval for that airplane (i.e., the airplane will be 'in transit' but will be 0km towards the destination airport).

Each airplane must wait at least 60 minutes (four intervals) after landing before taking off again, to allow for unloading, loading, and refueling. It may have to wait longer if other airplanes at the same airport are also ready to take off at the same time. Initially, all airplanes have been waiting at least 60 minutes, and so all are ready to take off.

The simulation decides where to send each airplane on a rotating ("round-robin") basis. In other words, a given airplane should be sent to the least-recently-used destination once it is ready for takeoff.

A airplane is given enough fuel to make it to its destination, plus another 15% in case it needs to circle at the destination airport. To calculate this, we must first know the time the trip will take:

$$\text{time} = \frac{\text{route distance}}{\text{airplane speed}}$$

Then we can calculate the fuel requirement:

$$\text{fuel} = \text{time} \times \text{airplane fuel consumption rate} \times 1.15$$

Airplanes have no particular take-off priority, provided they are ready to take off. If several airplanes are ready to take off at the same time, the simulation should select any one of them.

(Hint: if you add the airplanes to a queue when they land, then you can simply check the head of the queue to see if that airplanes is ready to take off.)

2.4 Flying

Airplanes must follow particular routes between airports, and each route is one-way. For instance, Perth→Melbourne is a different route from Melbourne→Perth, and the two may even have different distances due to different flight paths (to avoid head-on collisions).

Not all airports have routes between them. If there is no route from airport A to airport B, then an airplane cannot fly directly from A to B.

A given airplane always travels at a single specific "cruising speed", unless landed, circling or crashed. We will ignore acceleration, winds, and anything else that, in reality, would cause the speed to change. However, two airplanes may have different cruising speeds, and so it is possible for one to "overtake" another.

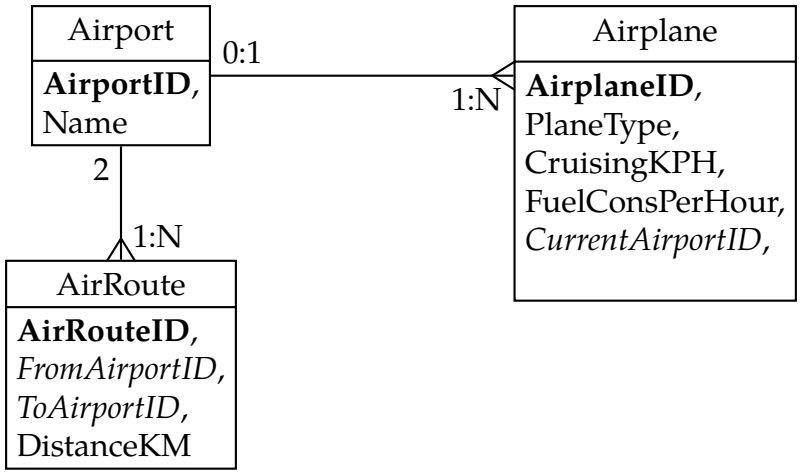
A given airplane always consumes fuel at a single specific rate, unless landed or crashed. (This includes while the airplane is circling at an airport, and so not actually covering any distance.)

2.5 Initial Database

You will have access to a database in the form of a DLL file called `ATCDatabase.dll`. This will contain information on the various airports, air routes and airplanes to be simulated.

This database exists purely to provide initial information to the simulation. There is no need to save data back to the database, and the database does not specify "what happens" in the simulation.

The database has three tables, structured as follows:



Each airport initially has:

- AirportID – a unique integer identifying the airport.
- Name – a text label for the airport; e.g. "Perth" (which is not the same as the ID!).

Each air route has:

- AirRouteID – a unique integer identifying the route.
- FromAirportID and ToAirportID – the IDs of the origin and destination airports.
- DistanceKM – the distance between the two airports in kilometres. (Note: the distance from airport A to airport B *can* be different to the reverse route from B to A, since the flight path might be different.)

Each airplane initially has:

- AirplaneID – a unique integer identifying the airplane.
- PlaneType – the type of plane (e.g., "Boeing 747").
- CruisingKPH – the speed at which the airplane flies (in km per hour). Note that (for simplicity) airplane *always* travel at exactly this speed unless landed.
- FuelConsPerHour – the amount of fuel used per hour when flying. (No fuel is used when landed.)
- CurrentAirportID – the ID of the airport at which the plane is currently landed. All planes are initially landed, but the simulation should set this to -1 while a plane is in-transit.

Note: `ATCDatabase.dll` is *not* a C/C++ DLL. Rather, it is written in C#, and hence much easier to "import" than what we did in the practicals. Specifically, you treat it exactly like you did the TrueMarbleData and simply do an Add Reference to the DLL. .NET will take care of the rest by automatically inspecting the DLL and determining its classes and methods. Then all you need to do is add the "`using ATCDatabase;`" clause and away you go. (.NET cannot do the same with C/C++ DLLs since they do not include .NET-required supporting functions in the DLL.)

2.6 Simulation-Updated Data

The simulation itself will generate additional data that must also be tracked. Specifically:

Each airport should *also* have:

- A queue of airplanes that are currently landed (where the head of the queue is the airplane that will take off next).
- A list of airplanes that are circling.
- A list of airplanes that are entering-circling.
- The next destination to send an airplane to (an AirportID). Each airport must know its outbound routes. When an airplane takes off for one destination, the "next destination" should be updated.

  (Note: an airport does not necessarily have a route to all other airports. You can only send an airplane to another airport if there is actually a route from the first to the second.)

Each airplane should *also* record:

- The route it's currently flying on (a AirRouteID). This should be -1 if the airplane is landed.
- The distance (in kilometres) that the airplane has travelled so far along its current route. Zero indicates that the plane is landed, or has only just taken off. This must be updated each time interval while the airplane is in transit or entering circling.
- The time the airplane has spent landed so far, or -1 if the airplane is in the air. This must be updated each time interval while the plane is landed.
- The remaining fuel. This must be updated each time interval the airplane spends in transit, entering circling, or circling. If the fuel runs out (drops to zero) in these states, then the airplane crashes. (Fuel is expressed as a unitless number.)

3 Presentation Tier

The application must allow the user to play the simulation one step at a time, and to view simulation data as the simulation progresses. You must also write *TWO* front-ends for this: a .NET WPF-based GUI, and a web GUI.

3.1 Windows WPF GUI

The GUI must have a "Step" button, which (when pressed) moves the simulation forward one time interval.

The GUI must also display a list of all airports, showing ID and Name. The user must be able to select an airport in order to view the status of all its associated airplanes.

When viewing airplanes, *two* lists must be shown:

- All inbound/arriving planes (entering circling, circling, crashed) and,
- All outbound/departing planes (landed or in-transit to another airport).

Lists should be displayed using something like a ListView or GridView. The GUI should display *all* aircraft information: ID, state, type, speed, fuel consumption rate, fuel remaining, route ID, distance along route and time landed.

How you wish to organize the GUI to present this information is up to you. A couple of possibilities include:

- Have a main window that lists airports and controls the playing of the simulation. When the user selects an airport to view, and the GUI pops up a dialog window to show the airplane status info at that airport. Whether you make the dialog window modal (simpler) or modeless (allows for viewing multiple airports at once) is up to you.
- Have a single large window split into two (or three) panes, with airports on the left and the two lists of airplane status info (for the currently-highlighted airport) on the right.
- Any other reasonably sensible layout!

3.2 Web GUI

In addition to the WPF GUI, you must also implement a Web version of the same GUI, though somewhat simplified:

- The main screen should show the list of airports, and a Step button to drive the simulation one 15-minute step forward.
- Airports should be selectable (e.g., putting a button beside each row, or similar) and when chosen should display a new page showing the airplanes controlled by that airport.
- Unlike the WPF GUI, when viewing the airplanes in an airport, the page only needs to show a single list containing *all* airplanes, rather than splitting inbound vs outbound planes.
- Furthermore, only the ID, state, type, and remaining fuel need to be shown for each aircraft.

Remember to use the IsPostback property in Web system to detect whether the page is being initially loaded or being posted back to from another page.

Note: Don't confuse `<asp:button>` with `<input type="button">`. The former requires an onclick attribute for calling C# server-side code whereas the latter uses onserverclick to do this.

4 Business Tier

The business tier must allow a client to connect to it and facilitate the client requesting that the simulation take a step in 'time' as well as requests for data (Airports, Airplanes, etc).

The business tier is actually split between two processes – a 'master' that controls the simulation and a set of 'slaves' that represent the ATC controllers. The master should be the one to contain the database and provide services to load the initial data objects from this (however, once loaded into objects the database is fairly irrelevant – do not update the database).

A time step of the simulation will involve the master requesting the ATC slaves to perform the appropriate updates of airplane positions and any hand-over from one airport to another. Hand-over (i.e., the transition of a plane from in-transit to entering-circling) would best be handled by having the slave ask the master to pass the airplane to the destination airport rather than directly between slaves.

The simulation time-step *must* be done in parallel: that is, the master must not wait for the first slave to complete their simulation step execution before firing the next slave up. See the Week 3 lecture notes on how to use blocking async calls to perform RPC parallel processing. (You do not need to use completion callbacks here – they are unnecessarily complicated for this purpose.)

The question remains on how the slaves and master will set up their communication. Server initialization should occur as follows:

- The master is the first server to be run. It must be a singleton, and creates a service object, waiting ready for clients to connect.
- Subsequently, the first ATC slave 'server' is started. Do not make this a literal WCF service, but make it a client of the master with a *callback* interface implementation so that the master can contact the slaves later when the user requests simulation runs.
- The slave thus should connect to the master and add itself (or more accurately, get the master to add its callback implementation object) to a list of slaves that the master stores as a member field.
- Upon adding the slave to its list, the master should return to the slave the next unallocated Airport ID (or perhaps entire Airport) that the master has not yet offloaded to a slave.
- The slave should then build the list of airplanes and air-routes attached to this airport. Whether this is *given* by the master along with the Airport or *requested* later by the slave depends on how you design your system.
- Once four slaves have connected (this is why it must be singleton), the master should not allow new slaves – there are only four airports! (you can avoid hard-coding the four by asking the database how many airports there are).

After all servers have connected, the master can begin serving GUI clients.

- From this point, things get more 'normal', as you are used to in the practicals.
- The GUI client just asks the Master for the information it requires (e.g., initially the list of airports, later when showing an airport the lists of inbound/outbound airplanes) and tells the master to take a step forward in the simulation – all at the requests of the user.
- To respond to these requests the master will pull out information from the slaves (since they know what airplanes are at their airport) or command the slaves to take a step forward in time.

While debugging your application, you may like to avoid running four separate ATC programs. Instead, 'fudge' it by having the one ATC project create four ATC slave callback objects and connect four times to the master. (This is only for debugging though.)

Some additional hints:

- Slaves and master must communicate back-and-forth, so they need a Duplex channel and thus a callback interface. Since RPC is client-server, it's easiest to have the (singleton) master as the 'server' and the multiple slaves connecting to it as duplex 'clients' implementing the callback interface. So in this case, the slaves are NOT literally services, but rather technically clients. However, they are activated by the master on the callback channel whenever the GUI requests a simulation timestep, and so logically the slaves behave like servers for the master. But since it is really peer-to-peer behaviour, the distinction is irrelevant and you make the slaves 'clients' because that is easiest to implement.
- Since the Master is a singleton, the connecting slaves will not trigger any special event on the master (e.g., the constructor won't be called). So you need to have a method that the slaves must explicitly call after connecting to the master's channel, something like "`AddSlave(IMasterCallback newSlave)`". This should add the slave to the master's list of slaves, and presumably give back the Airport that is to be allocated to the slave (and thus get it off the master's hands, since the Slave should own the airport it is allocated).
- When the slaves call the master to AddSlave (or whatever), bear in mind that you can't actually pass the slave object over the network (pass-by-ref isn't allowed by WCF remember). Instead, in the master code of AddSlave you have to use OperationContext.Current.GetCallbackChannel<..>() to retrieve the ref to the slave for storage into a list of slaves that the master can use later on to step the simulation/etc. So you may feel that AddSlave is not a good name for it, and perhaps InitialiseSlave or RegisterSlave may be a better name.
- Since the slaves require a duplex channel w/callback to the master, and the GUI client also connects to the same master interface, you will need to make the GUI client connect as a duplex with a callback. But since the GUI client never gets callbacks, just create a dummy class that implements the callback interface with empty functions to get it compiling. Obviously this is a bit ugly. The alternative is to have *two* different interfaces in the master (one for slaves and one for GUI clients) – although this is conceptually a nicer design you will find it gives you more trouble/work in the end. So I recommend stick with one interface for the master and have the GUI clients implement a dummy callback class for the duplex channel.
- You may need to resolve race conditions, since the Master process is a singleton, and two or more clients can connect and try stepping the one-and-only simulation forward in time simultaneously. So you will need to solve the race condition with synchronizing appropriate methods (with the [MethodImpl(...)] attribute). Probably only the master needs this synchronizing since it is the 'bottleneck point', but you can be safe and do synchronisation on the slaves do. Only methods that update the simulator's state need to be synchronized (as race conditions only happen when *changing* data).

5 Design Hints

5.1 Design Process

Design is iterative. You will need to sketch out an idea of the classes you will need before hand (perhaps on paper). However, as you begin coding, you will undoubtedly discover that you need to change your design.

Therefore, don't draw up a polished UML class diagram right away. Wait until you've finished your code, and *then* draw the final versions of your diagrams.

5.2 Data Structures

When building your classes for storing info about the Airports, AirRoutes and Airplanes, it is best to view these classes as raw data storage (with a constructor attached for your convenience). Try to avoid putting in processing logic into them because these objects will be passed around *by value*, and there is no guarantee that the remote end can support the code that the object will be trying to execute.

Instead, put processing logic into the service (controller) classes: the master and the ATC slaves.

This may mean you have to load up secondary objects from the master all the time (e.g., for the GUI to display the name of an airport that an Airplane is landed at, you need to load the Airport as well), but that's how WCF works.

Put the .cs files of the data structure classes in the master project – every project references the master so it is a convenient central repository for the class definitions. Alternatively, make a separate DLL project whose sole purpose is to define the data structures, and include this DLL in all projects.

5.3 Ownership of Data Objects

Since copies of data objects are being passed around (by value), it can become confusing as to who has the 'authoritative' copy.

In this system, the answer isn't too difficult but you need to keep it in mind:

- Each ATC slave 'owns' the airport it is associated with
- The airport who is currently controlling/tracking an airplane 'owns' that airplane: anyone else just has a copy that can be used for its data, but shouldn't be updated.
- The AirRoute is a special case, being 'owned' by the two airports that it connects. Since air-routes don't get changed, this isn't a big deal.

5.4 Exception Handling

Don't forget to handle exceptions!

6 Report

In addition to your code, you must provide a professionally presented report (as a PDF) that incorporates the following items:

(a) **Instructions for running your application.**

You must briefly (but precisely!) explain how to run the application.

(b) **Known issues.**

State what parts of this assignment specification you have successfully implemented, to the best of your knowledge. Describe any known bugs, crashes, etc., and any workarounds you may have for them.

(c) **A deployment diagram.**

This must show how the system can be deployed in terms of server machine(s), client machine(s), and network/Internet links (distinguish between these!). You must also show where the boundaries between tiers occur.

(d) **A UML class diagram.**

This must show the main entities of your design and how they relate to each other. Include the methods and fields/properties of each class (but don't worry about showing parameters or overloaded versions, since that just causes clutter). Again, you must show where the boundaries between tiers occur. It must be clear which tier each class exists in.

Further notes:

- If a class inherits from .NET classes, show the inheritance but do not show the details of the .NET class (i.e., just put the name of the .NET class in).
- You may use Visual Studio's UML diagramming tool to simplify the process of producing the UML, but you will need to update it to show all message passing and other relationships that it cannot extract automatically.
- If the UML diagram is too large to fit on one page, place each tier in its own page. For relationships with classes in other tiers, just show the names of these cross-tier classes (since their full definition exists on the page for the other tier).

(e) **Three UML sequence diagrams.**

These must describe how the following interactions work:

- Slave start-up and allocation/initialisation of airport (for a single slave). This should cover slave construction, connection to master, allocation of the airport to the slave, and loading of the airport info (including controlled airplanes) by/for the slave.
- Taking one simulation step using the WPF client interface. This should cover everything from when the user clicks on 'Step', through parallel execution of the slaves and up to when the simulation step completes and returns to the client.

  You should indicate where the parallel execution of the slaves occurs. I am

not fussed how you wish to show this, but it must be made clear (e.g., explicitly indicating BeginInvoke calls).

- Displaying the airplanes in an Airport using the Web interface. This should cover everything from when the user clicks on the airport to view, through to loading the airplanes in the airport, building the list and returning the airplanes list page.

  Include the "onclick" button event handler, and any other significant code in your browser (e.g., Ajax).

As with the other diagrams, you must show where the boundaries between tiers occur. (Different objects may belong to different tiers.)

Each sequence diagram must show the methods called, the order they are called in, and the objects they are called on. You *do not* need to show parameter information.

You may omit certain calls from your sequence diagrams:

- Omit calls that happen outside your design. When you call .NET functions or ATCDatabase, the method you call will often make some calls of its own. You may omit these additional calls (assuming you even find out what they are), because they have nothing directly to do with your code.
- Omit *minor* .NET method calls that aren't important in understanding the flow of events (e.g., calls to `Convert`, setting GUI text fields, etc.).
- Do not omit *important* .NET calls; i.e., where they have a significant effect on the flow of events. They include functions/methods involving user interaction, asynchronous calls, completion callbacks, major GUI updates, etc.
- If in doubt about whether to include a particular call, *do* include it.

Drawing sequence diagrams can be tricky without a proper tool. If you search online for "sequence diagram drawing tool", you should find several good options. One of them, for instance, is www.websequencediagrams.com.

WARNING: You must not "auto-generate" your sequence diagrams, and you will get zero marks for them if you do! Some tools (like Visual Studio) can do this, but the resulting diagrams will have too many low-level details, and will not cross tier boundaries, so remote calls are "black boxes" without any detail. (You can still use Visual Studio to help create UML class diagrams though.)

(f) **Design essay.**

This should be at least two pages (single-spaced, 12pt, 2.5cm margins). It should discuss the pros and cons of your design choices, regarding the following:

- Network efficiency / bandwidth usage;
- GUI responsiveness;
- Design and implementation elegance / simplicity;
- Design effects of having data classes without significant processing;
- Issues in design due to restrictions on object passing;
- Effective use of distributed resources (e.g., split of processing load);
- Memory usage on servers and clients;
- Any other design aspects.

Structure your discussion so that you examine the trade-offs you have made in aspects of your system. You must consider how well your system will cope in difficult circumstances, such as low-bandwidth networks, dozens of client PCs simultaneously connected, etc.

Note: there *are* disadvantages to whatever design approach you choose to take, and it is critical that you know what these are. You are not expected to design a perfect system (because that's impossible), but you *are* expected to be able to articulate the imperfections, and to justify them in terms of alternate design possibilities.

7 Submission

Submit a single .zip or .tar.gz file containing all your files (projects, report, etc.). Submit your entire assignment electronically, via Blackboard (lms.curtin.edu.au), before the deadline.

Submit one .zip or .tar.gz file containing:

A declaration of originality – whether scanned or photographed or filled-in electronically, but in any case *complete*.
Your Visual Studio project(s) – including all your source code.
Your report – a single PDF file containing everything mentioned in Section 6.

Note: do not use .zipx, .rar, .7z, etc.

You are responsible for ensuring that your submission is correct and not corrupted. You may make multiple submissions, but only your newest submission will be marked.

Late assignment submissions *will not be accepted* (as per the Unit Outline).

8 Academic Misconduct – Plagiarism and Collusion

This is an assessable task, and as such there are strict rules. You must not ask for or accept help from *anyone* else on completing the tasks. You must not show your work to another student enrolled in this unit who might gain unfair advantage from it.

These things are considered **plagiarism** or **collusion**.

Staff can provide assistance in helping you understand the unit material in general, but nobody is allowed to help you solve the specific problems posed in this document. The purpose of the assignment is for *you* to solve them *on your own*.

Please see Curtin's Academic Integrity website for information on academic misconduct (which includes plagiarism and collusion).

The unit coordinator may require you to provide an oral justification of, or to answer questions about, any piece of written work submitted in this unit. Your response(s) may be referred to as evidence in an Academic Misconduct inquiry. In addition, your assignment submission may be analysed by Turnitin and/or other systems to detect plagiarism and/or collusion.

9 Marking Criteria

Your submission will be marked as follows:

Criterion	Marks
Architectural Design (Deployment and Class Diagrams)	
• Coverage of user requirements	4 marks
• Tiers shown and labelled appropriately	2 marks
• Deployment architecture diagram	5 marks
Subtotal	**11 marks**
Detailed Design (Sequence Diagrams and Design Essay)	
• UML Sequence diagram: Slave start-up	4 marks
• UML Sequence diagram: WPF sim step	5 marks
• UML Sequence diagram: Web display airport	5 marks
• Penalty for auto-generating diagrams from code	–100% off relevant items
• Design essay	10 marks
Subtotal	**24 marks**
Implementation (Code)	
• Good commenting	4 marks
• Clarity	4 marks
• Consistency	3 marks
• Coverage of user requirements	
– Simulation implementation	20 marks
– Async slave processing	2 marks
– WPF simulation/airport display	7 marks
– WPF airplane listing	10 marks
– Web simulation/airport display	10 marks
– Web airplane listing	5 marks
• Class differs from UML design	up to -3 marks per class
• Not a distributed system	up to -100% this section
• Use of Web Binding / LINQ	up to -30% this section
Subtotal	**65 marks**
Total	**100 marks**

End of Assignment