Practical six: The Actor pattern

Wrap up

epcc

THE UNIVERSITY OF EDINBURGH

The problem

epcc

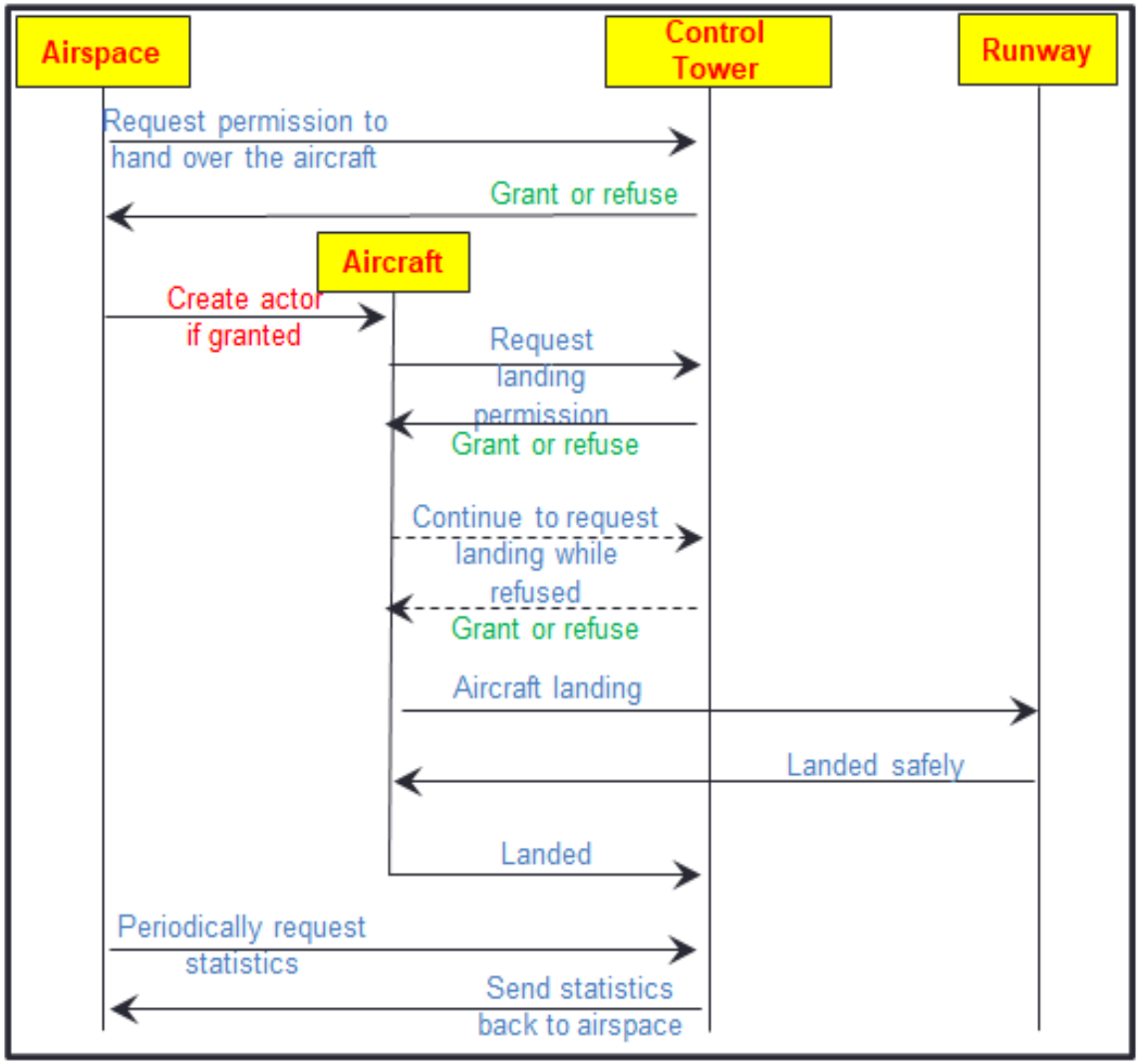
THE UNIVERSITY OF EDINBURGH

Use the actor pattern to model this via MPI

- Four types of actor
  - Air traffic control tower
  - Runway
  - Airspace (the operator introducing/handing aircraft to the ATC tower)
  - Aircraft

- Control tower, runway and airspace are created once at model start up and exist till the end
- Aircraft are much more dynamic, created as the model runs and many actors can be created (and can die)

Interaction pattern

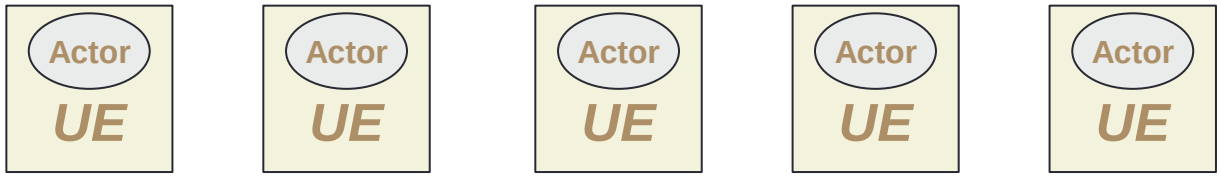


epcc

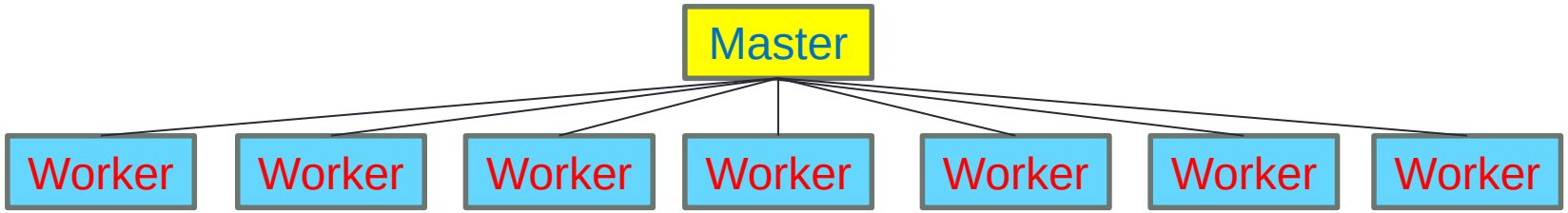
THE UNIVERSITY OF EDINBURGH

The plan…

- One actor per UE as much simpler to do



- Using the same process pool as last weeks practical



Function	Description
int processPoolInit()	Initialises the process pool (1=worker, 2=master)
void processPoolFinalise()	Finalises and process pool (called from all)
int masterPoll()	Master polls to determine whether to continue or not
int workerSleep()	Worker waits for new task (1=new task, 0=stop)
int startWorkerProcess()	Starts a new worker task and returns the rank of this
int getCommandData()	Retrieves the rank of the task created this one
void shutdownPool()	Called by anyone to shut down the pool

epcc

THE UNIVERSITY OF EDINBURGH

Initialisation

```
int statusCode = processPoolInit();
if (statusCode == 1) {
    workerCode();
} else if (statusCode == 2) {
    // Create initial actors
    int masterStatus = masterPoll();
    while (masterStatus) {
        masterStatus=masterPoll();
    }
}
processPoolFinalise();
```

```
int statusCode = processPoolInit();
if (statusCode == 1) {
    workerCode();
} else if (statusCode == 2) {
    createInitialActor(0);
    createInitialActor(1);
    createInitialActor(2);
    for (i=0;i<INITIAL_AIRCRAFT;i++) {
        createInitialActor(3);
    }
    int masterStatus = masterPoll();
    while (masterStatus) {
        masterStatus=masterPoll();
    }
}
processPoolFinalise();

static void createInitialActor(int type) {
    int data[1];
    int workerPid = startWorkerProcess();
    data[0]=type;
    MPI_Bsend(data, 1, MPI_INT, workerPid, 0, MPI_COMM_WORLD);
}
```

```
static void workerCode() {
    int workerStatus = 1;
    while (workerStatus) {
        int parentId = getCommandData();
        // Your job to complete
        workerStatus=workerSleep();
    }
}
```

```
static void workerCode() {
    int workerStatus = 1, data[1];
    while (workerStatus) {
        int parentId = getCommandData();
        MPI_Recv(data, 1, MPI_INT, parentId, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        if (data[0] == 0) {
            airspace(INITIAL_AIRCRAFT);
        } else if (data[0] == 1) {
            control_tower(INITIAL_AIRCRAFT);
        } else if (data[0] == 2) {
            runway();
        } else if (data[0] == 3) {
            aircraft();
        }
        workerStatus=workerSleep();
    }
}
```

epcc

THE UNIVERSITY OF EDINBURGH

Airspace actor

This actor is both driven by messages and also by time (background behaviour to monitor this)

```
static void airspace(int initialAircraft) {
    int failed_passes=0, stat_data[2];
    int new_ac_data[2], atcdata, outstanding_pass_over=0, elements, stat_day;
    int outstanding_retrieve_stats=0, finish_actor=0, outstanding,  total_aircraft=initialAircraft;
    MPI_Status status;
    struct timeval curr_time;
    gettimeofday(&curr_time, NULL);
    time_t seconds=0, start_seconds=0, pass_ac_ms=0, ms;
    start_seconds=curr_time.tv_sec;
    pass_ac_ms=curr_time.tv_sec*1000 + curr_time.tv_usec/1000;
    while (1==1) {
        gettimeofday(&curr_time, NULL);
        ms=curr_time.tv_sec*1000 + curr_time.tv_usec/1000;
        if (ms - pass_ac_ms > 250) {
            pass_ac_ms=ms;
            if (!outstanding_pass_over) {
                total_aircraft++;
                atcdata=PASS_AIRCRAFT_OVER;
                MPI_Bsend(&atcdata, 1, MPI_INT, CTRL_TOWER_ACTOR_RANK, 0, MPI_COMM_WORLD);
                outstanding_pass_over=1;
            }
        }
        if (curr_time.tv_sec != seconds) {
            seconds=curr_time.tv_sec;
            if ((seconds - start_seconds) % DAY_LENGTH == 0) {
                stat_day=(seconds - start_seconds) / DAY_LENGTH;
                atcdata=RETRIEVE_STATISTICS;
                MPI_Bsend(&atcdata, 1, MPI_INT, CTRL_TOWER_ACTOR_RANK, 0, MPI_COMM_WORLD);
                outstanding_retrieve_stats=1;
            }
            if ((seconds- start_seconds) > (DAY_LENGTH*MAX_DAYS)) {
                atcdata=FINISH;
                MPI_Bsend(&atcdata, 1, MPI_INT, CTRL_TOWER_ACTOR_RANK, 0, MPI_COMM_WORLD);
                finish_actor=1;
            }
        }
        if (!outstanding_pass_over && !outstanding_retrieve_stats && finish_actor) {
            break;
        }
        MPI_Iprobe(CTRL_TOWER_ACTOR_RANK, 0, MPI_COMM_WORLD, &outstanding, &status);
        if (outstanding) {
            MPI_Get_count(&status, MPI_INT, &elements);
            if (elements == 1) {
                outstanding_pass_over=0;
                MPI_Recv(&atcdata, 1, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                if (atcdata == PERMISSION_GRANTED) {
                    int workerPid = startWorkerProcess();
                    new_ac_data[0]=3;
                    MPI_Bsend(new_ac_data, 1, MPI_INT, workerPid, 0, MPI_COMM_WORLD);
                } else {
                    failed_passes++;
                }
            } else if (elements == 2) {
                MPI_Recv(stat_data, 2, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                printf("Status at day %d\nTotal number of aircraft: %d, Number of successful landings: %d, Number aircraft refused %d, Number being handled by ATC: %d\n\n",
                    stat_day, total_aircraft, stat_data[0], failed_passes, stat_data[1]);
                outstanding_retrieve_stats=0;
            } else {
                printf("Airspace has got %d elements and is not sure how to handle this\n", elements);
            }
        }
    }
}
```

Every 250ms request to control tower the passing over of a new aircraft (add aircraft to simulation)

Periodically (every day) request statistics from control tower

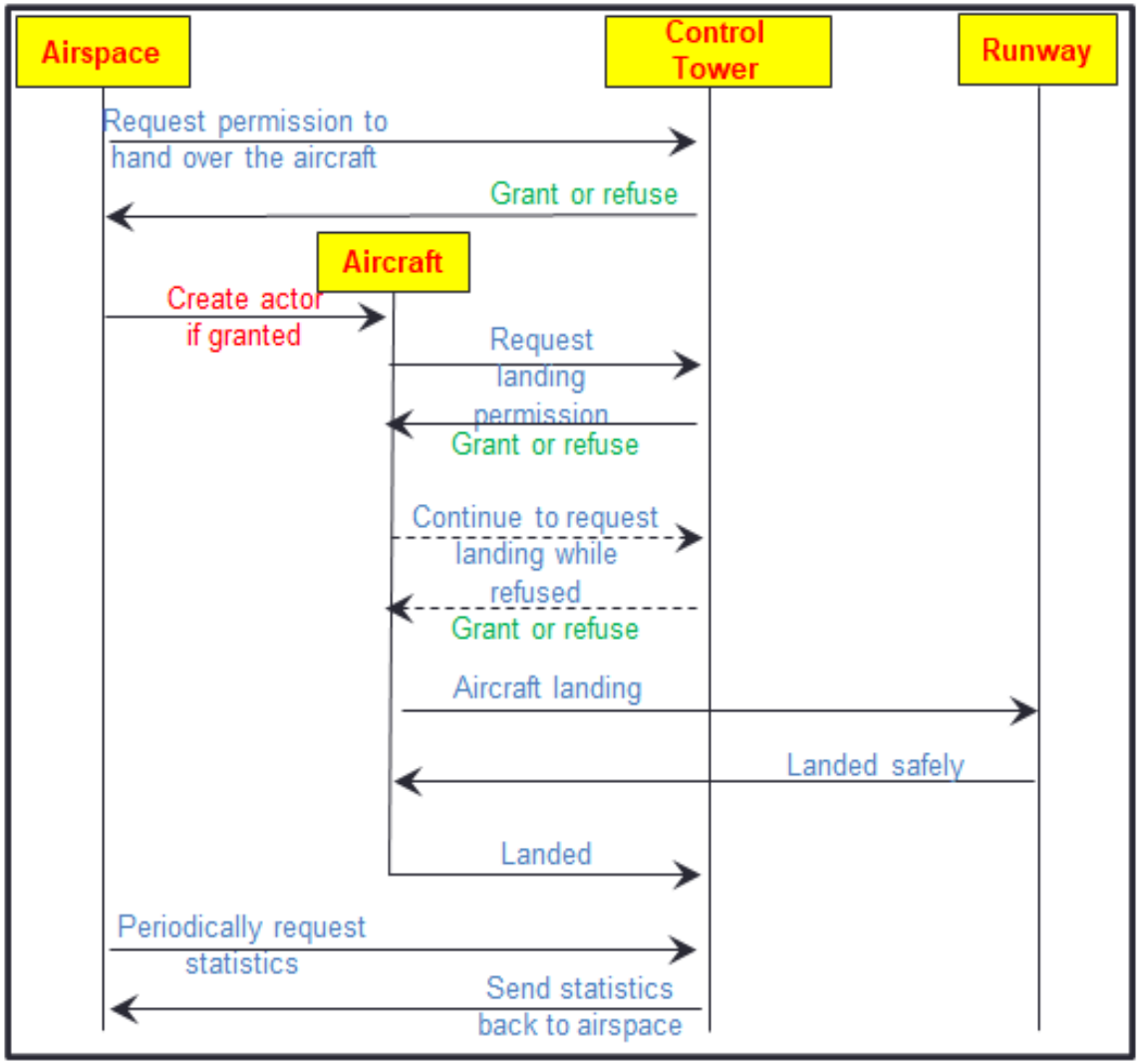
If the maximum number of days is up, then tell the control tower about terminations

With both of these are expecting a reply from the control tower, so set a flag

Probe for data

Based on the number of elements sent to this by the control tower we know if it's permission granted or refused for the aircraft transfer, or statistics

Interaction pattern



epcc

THE UNIVERSITY OF EDINBURGH

Control tower actor

This actor is entirely driven by messages

```
static void control_tower(int initialAircraft) {
    int data, permission_data, stat_data[2];
    int runway_busy=0, successful_landings=0, current_aircraft_number=initialAircraft, finish_actor=0;
    MPI_Status status;
    while (1==1) {
        MPI_Recv(&data, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
        if (data == LAND_REQUEST) {
            if (!runway_busy) {
                runway_busy=1;
                permission_data=PERMISSION_GRANTED;
            } else {
                permission_data=PERMISSION_REFUSED;
            }
            MPI_Bsend(&permission_data, 1, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD);
        } else if (data == LANDED) {
            runway_busy=0;
            successful_landings++;
            current_aircraft_number--;
        } else if (data == PASS_AIRCRAFT_OVER) {
            if (current_aircraft_number < MAX_NUMBER_CONCURRENT_AIRCRAFT) {
                current_aircraft_number++;
                permission_data=PERMISSION_GRANTED;
            } else {
                permission_data=PERMISSION_REFUSED;
            }
            MPI_Bsend(&permission_data, 1, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD);
        } else if (data == RETRIEVE_STATISTICS) {
            stat_data[0]=successful_landings;
            stat_data[1]=current_aircraft_number;
            MPI_Bsend(stat_data, 2, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD);
        } else if (data == FINISH) {
            finish_actor=1;
        }
        if (finish_actor && runway_busy==0 && current_aircraft_number==0) {
            shutdownPool();
            break;
        }
    }
}
```

Landing request, keeps track of whether runway is busy or not and sends permission based on this back to the aircraft

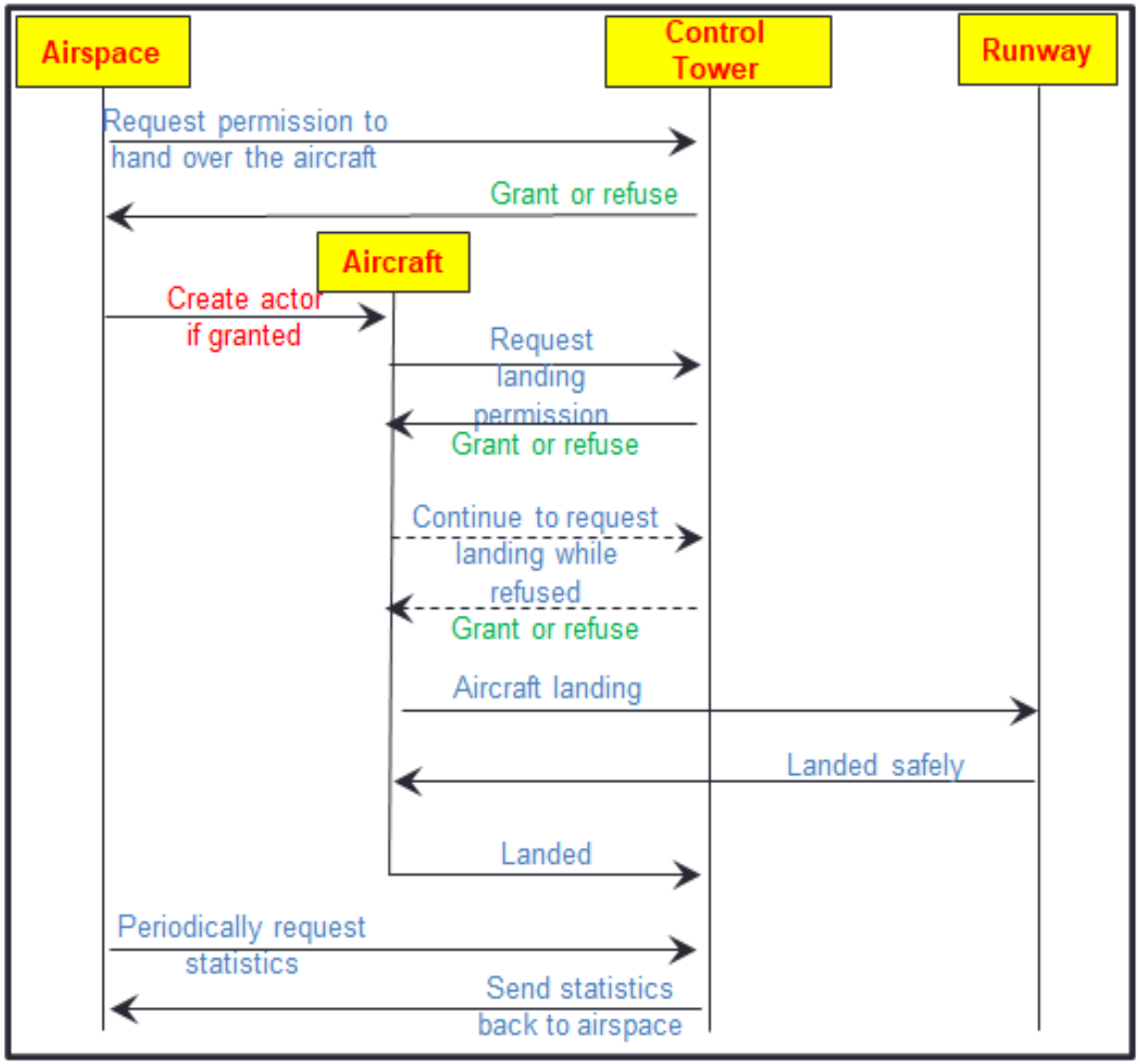
When aircraft lands mark runway is free and increment statistics counter

Request from airspace actor to hand off an aircraft (effectively add a new actor to the system)

Request from airspace to retrieve statistics

Termination

Interaction pattern



epcc

THE UNIVERSITY OF EDINBURGH

Runway actor

This actor is entirely driven by messages

```
static void runway() {
    int data, ack=LANDED, outstanding;
    MPI_Status status;
    while (1==1) {
        MPI_Iprobe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &outstanding, &status);
        if (outstanding) {
            MPI_Recv(&data, 1, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            if (data == LANDING) {
                MPI_Bsend(&ack, 1, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD);
            }
        }
        if (shouldWorkerStop())  break;
    }
}
```

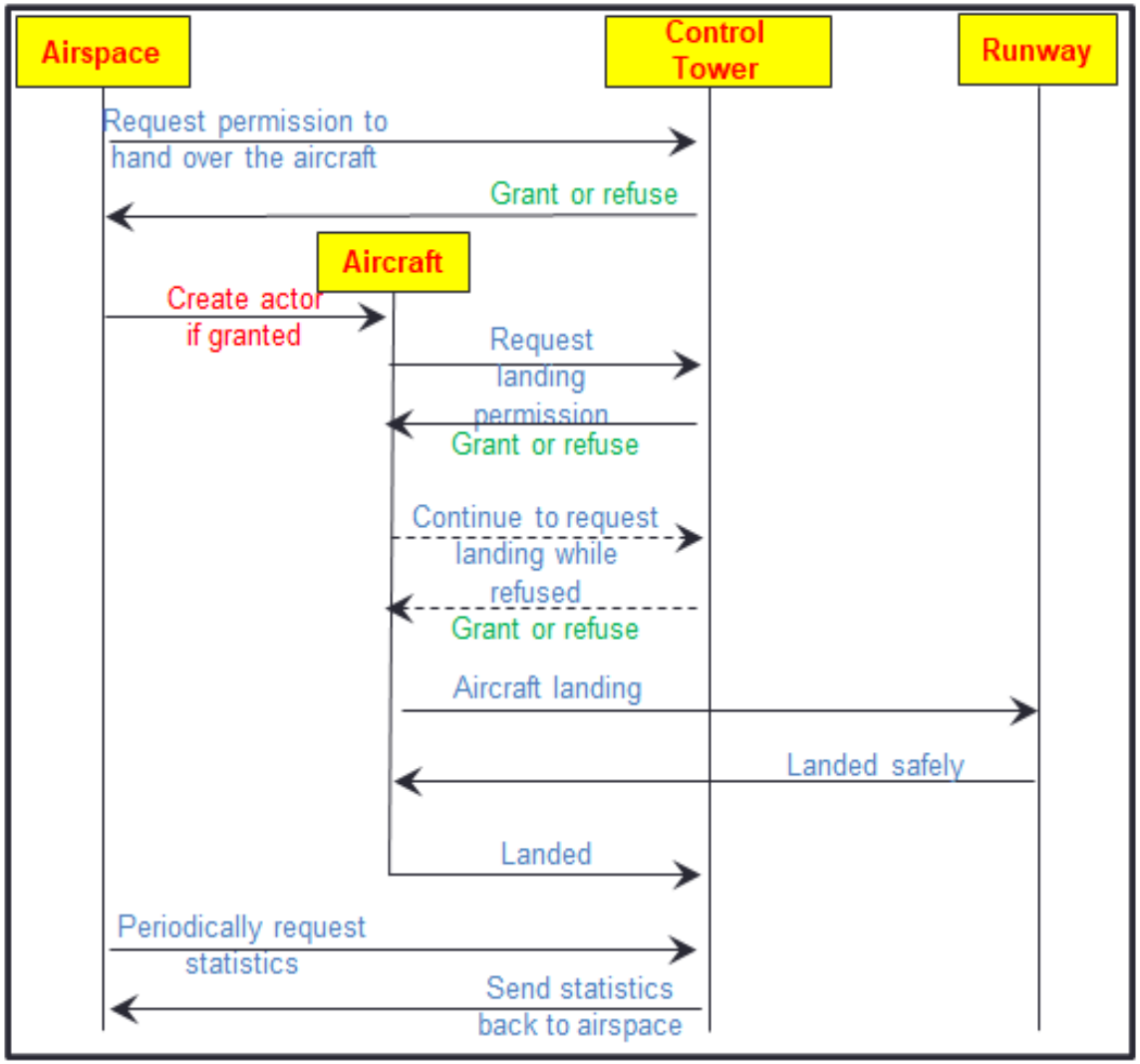
This actor is very simple, just driven by messages from aircraft and immediately acknowledges them

- *But still needs to check periodically whether the worker should stop (e.g. process pool has been terminated)*

epcc

THE UNIVERSITY OF EDINBURGH

Interaction pattern



epcc

THE UNIVERSITY OF EDINBURGH

Aircraft actor

This actor is drives itself and is not really driven by messages

```
static void aircraft() {
    int data=LAND_REQUEST, returnedAck;
    MPI_Send(&data, 1, MPI_INT, CTRL_TOWER_ACTOR_RANK, 0, MPI_COMM_WORLD);
    MPI_Recv(&returnedAck, 1, MPI_INT, CTRL_TOWER_ACTOR_RANK, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    while (returnedAck == PERMISSION_REFUSED) {
        sleep(1);
        MPI_Bsend(&data, 1, MPI_INT, CTRL_TOWER_ACTOR_RANK, 0, MPI_COMM_WORLD);
        MPI_Recv(&returnedAck, 1, MPI_INT, CTRL_TOWER_ACTOR_RANK, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    data=LANDING;
    MPI_Bsend(&data, 1, MPI_INT, RUNWAY_ACTOR_RANK, 0, MPI_COMM_WORLD);
    MPI_Recv(&returnedAck, 1, MPI_INT, RUNWAY_ACTOR_RANK, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    data=LANDED;
    MPI_Bsend(&data, 1, MPI_INT, CTRL_TOWER_ACTOR_RANK, 0, MPI_COMM_WORLD);
}
```

Request landing from tower

While permission is refused re-request each second

Tell runway is landing and wait for acknowledgement

Tell control tower has landed

- This is simple as much of the behaviour is sending a message and receiving some sort of acknowledgement.

epcc

THE UNIVERSITY OF EDINBURGH

The challenge of termination…..

- Termination can be a big challenge here, especially with complex systems

```
if (!outstanding_pass_over && !outstanding_retrieve_stats && finish_actor) {
    break;
}
```

Airspace actor will simply break out of receive loop when flags are met

```
if (finish_actor && runway_busy==0 && current_aircraft_number==0) {
    shutdownPool();
    break;
}
```

Control tower actor actually shuts the process pool down

```
static void runway() {
    int data, ack=LANDED, outstanding;
    MPI_Status status;
    while (1==1) {
        MPI_Iprobe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &outstanding, &status);
        if (outstanding) {
            MPI_Recv(&data, 1, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            if (data == LANDING) {
                MPI_Bsend(&ack, 1, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD);
            }
        }
        if (shouldWorkerStop())  break;
    }
}
```

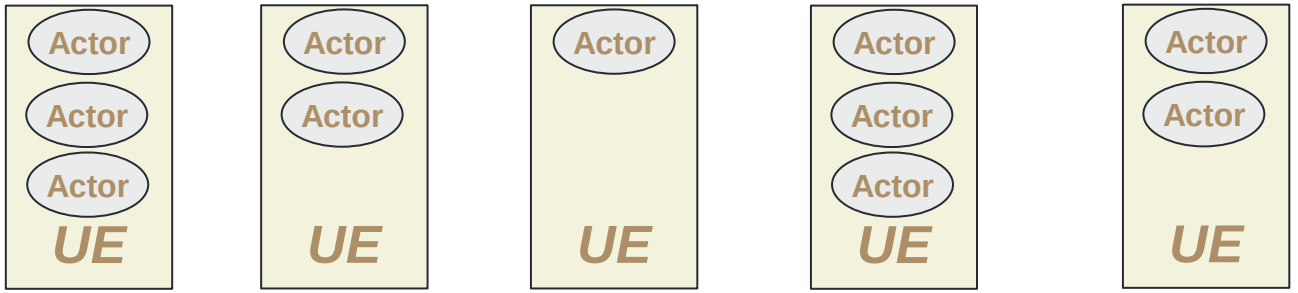
Runway actor periodically checks whether shutdown has happened

- Aircraft actor never checks termination criteria at all
- Must be absolutely sure that messages match, for instance an actor is not waiting on a message from another one which has shut down
- Tends to be driven by model logic and can be a source of deadlock here
- We use these flags here as part of this

epcc

THE UNIVERSITY OF EDINBURGH

A note on running multiple actors per UE



- This will decouple the number of actors from the UEs, and potentially give us better overall resource usage especially as the actors are trivial here
  - However adds lots of complexity, for instance have to decode messages to determine exactly which actor on a UE it belongs to
  - Need some form of co-operative multi-tasking
    - When waiting for a response pause and give another actor some time on the UE

epcc

THE UNIVERSITY OF EDINBURGH