Introduction to Game Programming and Robotics

Unit # 5

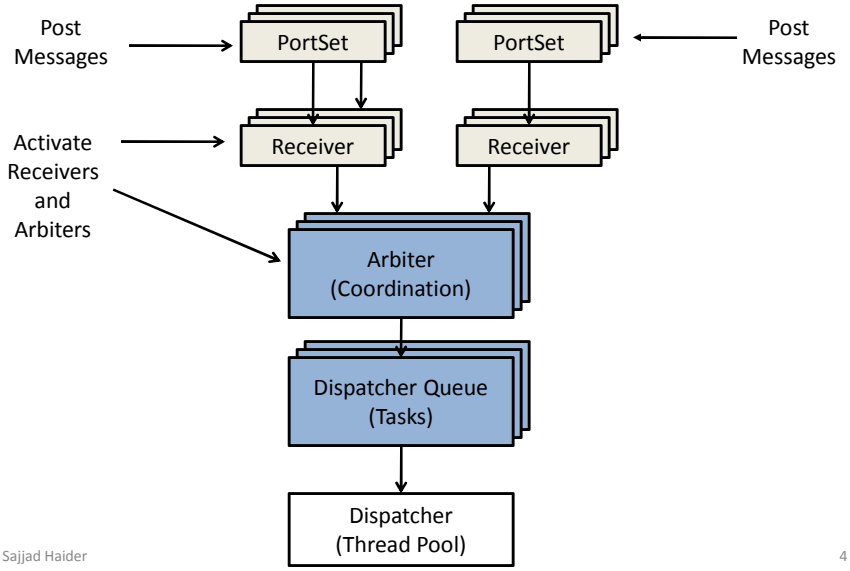
Concurrency and Coordination Runtime (CCR)

- CCR makes it possible to write segments of code that operate independently.
- It communicates by passing *messages*. When a message is received, it is placed in a queue, called *port*, until it can be processed by *receiver*.
- A *causality* holds a reference to a port that is used to report errors.

Decentralized Software Services (DSS)

- The CCR enables segments of code to pass messages and run in parallel within a single process.
- The DSS library extends this concept across processes and even across machines.
- An application built with DSS consists of multiple independent services running in parallel.
- It requires different mindset to write software using CCR and DSS.

CCR Architecture



Difference Between Dispatcher & Dispatcher Queue

- When you are in a queue to get your boarding pass at an airport, you are in a Dispatcher Queue.
- When you are asked to go to a particular counter to get your boarding pass, you are dispatched to a dispatcher.
- Similar examples – Barber Shop, Banks, etc.

Working of Spawn

Spawn Method

Microsoft Robotics Class Reference

Overload List

	Name	Description
	Spawn(Handler)	Invoke a message handler asynchronously, supplying any arguments explicitly.
	Spawn(T0)(T0, Handler(T0))	Invoke a message handler asynchronously, supplying any arguments explicitly.
	Spawn(T0, T1)(T0, T1, Handler(T0, T1))	Invoke a message handler asynchronously, supplying any arguments explicitly.
	Spawn(T0, T1, T2)(T0, T1, T2, Handler(T0, T1, T2))	Invoke a message handler asynchronously, supplying any arguments explicitly.

Spawn Method (Handler)

Microsoft Robotics Class Reference
Invoke a message handler asynchronously, supplying any arguments explicitly.

Namespace: Microsoft.Ccr.Core
Assembly: Microsoft.Ccr.Core (in Microsoft.Ccr.Core.dll) Version: 2.1.9999.0

Syntax

C#

```
protected void Spawn(
    Handler handler
)
```

Example 1 - Revisited

- Add Console.readline() in Example 1 code.
- This blocks the thread attached with the main() program and the other calls to SimpleHandler() function have to managed with the remaining threads in the pool.

Creating Your Own Dispatcher

- Another way to get overlapped behavior is to create your own dispatcher so you can specify the number of threads explicitly.
- In the CCR you need to *activate* a task to get it to execute, which means enqueuing it to a dispatcher queue.

Dispatcher - Example

```
void RunFromHandler()
{
    // Explicitly create a Dispatcher so we can control the pool size
    Dispatcher d = new Dispatcher(3, "Test Pool");
    DispatcherQueue q = new DispatcherQueue("Test Queue", d);

    // Activate FOUR tasks with a pool of THREE threads
    Arbiter.Activate(q,
       Arbiter.FromHandler(SimpleHandler),
       Arbiter.FromHandler(SimpleHandler),
       Arbiter.FromHandler(SimpleHandler),
       Arbiter.FromHandler(SimpleHandler));
}
```


Ports and Messages

- The most important class in the CCR is *Port*, which is basically a First-In-First-Out (FIFO) queue for *messages*.
- Messages, also referred to as *requests* or *responses*, are just objects of a specified type.
- Example
  - Port<int> intPort = new Port<int>();
  - intPort.Post(50);
- This code creates a new port called intPort, and sends the value 50 to it as a message using the *Post* method.

Ports – Creation and Post

- Because intPort is of type int, you can only *post* (enqueue) integer values to it.
- The posted value of 50 remains in the port queue until it is dequeued either by explicitly read or by a *receiver*.

Port - Purpose

- The major advantage of the port is that if you have a handle to a port – for example, if it is a global variable – then you can post messages to it from any thread and it will always be a safe operation.
- Furthermore, if all the receivers are busy, the message simply waits until it can be processed. The sender of the message does not have to wait because posting does not block.

Reading from a Port

- To see what the port contains (for debugging purposes), you can use the *ToString* method:
  - Console.WriteLine(intPort.ToString());
- A port has an ItemCount property that can be used to determine how many items are queued.
- However, if you want to see whether an item is available and retrieve it (if there is one), you can use the Test method, which returns the item as an object, or null if there is no item available.

```
int j ;
If (intPort.Test(out j))
    Console.WriteLine("Got Value : " + j);
Else
    Console.Writeline("No Value");
```


Port - Example

- Example on Page 52.

Port - ReceiveThunks

- It is worth pointing out here that the number of Receive Thunks (a thunk is a chunk of receiver code — that is, the handler in a task) is zero in all cases, and nothing is listed under Receive Arbiter Hierarchy .
- This is because no receivers are specified in this example.

Receivers and Arbiters

- Strictly speaking, a port has a second queue, which is a list of *receivers* to be executed to retrieve and process messages.
- The *Choice* arbiter, which you can think of as a logical *OR*, waits on two recievers until one of them fires. It then shuts down the unused receiver.
- Similar to a logical *AND*, a *JoinReceive* arbiter waits for two receivers to complete before continuing.

Continuation

- These receivers are sometimes called *continuations* and are conceptually similar to callback procedures in that they execute asynchronously when some event occurs — in this case the arrival of a message (or multiple messages).
- Much of the coding in MRDS services involves building receivers and the handlers that execute when messages are received.

Receiver - Example

- You have already seen how messages are posted and can be extracted using Test . Now consider the case of a receiver.
- To use a receiver, you must construct it and then attach it to the port.

```
Activate(
    Arbiter.Receive(false, intPort,
      delegate(int n)
                {Console.WriteLine("Receiver 1 " + n.ToString()); }
    )
) ;
```

- The Arbiter.Receive constructs a receiver using the following parameters:
  - A persistent flag, which is false in the preceding code, i.e., non - persistent — it only receives once and then is removed
  - A port called intPort to read from
  - A delegate that is executed in - line

Receiver – Application to Robots

- If the first parameter is true , then the receiver is persistent and will remain in the port ' s receiver list after processing a message (unless it is explicitly removed later).
- Many of the receivers that you create to control a robot need to be persistent.