Introduction to Game Programming and Robotics

Unit # 4

Statistics (Source: World Robotics 2008)

- With 12,000 units, **service robots** in defense, rescue and security applications accounted for 25% of the total number of service robots for professional use installed up to the end of 2007.
- They are followed by
  - field robots (mainly milking robots) with 20%,
  - cleaning robots and underwater systems with 12% each.
  - construction and demolition robots (9%),
  - medical robots (9%) and
  - mobile robot platforms for general use (7.4%) come in the next ranges.

Statistics (Source: World Robotics 2008)

- The total value of service robots for professional use installed up to the end of 2007 was about US$7.8 billion.
- The most expensive robots are
  - underwater systems,
  - medical robots,
  - milking robots and
  - robots for defense, rescue and security applications.

Statistics (Source: World Robotics 2008)

- Service robots for personal and private use (till 2007)
  - about 3.4 million units for domestic use and
  - about 2.0 million units for entertainment and leisure sold up to end 2007
- About 3.3 million vacuum cleaner and more than 110,000 lawn mowers were sold. The total value amounted to about US$1.3 billion.
- Projections for the period 2008-2011
  - about 12.1 million units of service robots for personal use to be sold
  - about 3.2 million robots for education and training are expected to be sold.

Need for Robotics Programming

- The market for these domestic robots will continue to grow, and there will soon be an urgent need for developers with knowledge of robotics programming.

Robotics at Microsoft (Wish list collected by Tandy Trower)

- Easily configure sensors and actuators and be able to run them asynchronously
- Start and stop software components dynamically
- Monitor the robot interactively and as it is operating
- Allow more than one person to access a single robot or allow one person to access multiple robots
- Reuse software components across robots

CCR and DSS

- MRDS is built on two basic components: **CCR** (Concurrency and Coordination Runtime) and **DSS** (Decentralized Software Services).
- The CCR is a programming model for handling multi-threading and inter-task synchronization, whereas DSS is used for building applications based on a loosely coupled service models.
- Services can run anywhere on the network.

Need for Concurrency

- At a higher level, MRDS is similar to an operating system for robots.
- CCR eliminates many of the issues related to multi-threading.
- It uses its threading mechanism, which is much more efficient than the Windows threading model.
- Another issue with robots is that events happen asynchronously. You don't want your service to be constantly pooling the robot for sensor information.
- The CCR supplies the underlying infrastructure that enables multiple tasks to execute concurrently on a single computer.

More on CCR

- CCR is a light weight library that is supplied as a .NET DLL.
- It is designed to handle asynchronous communication between loosely coupled services that are running concurrently.

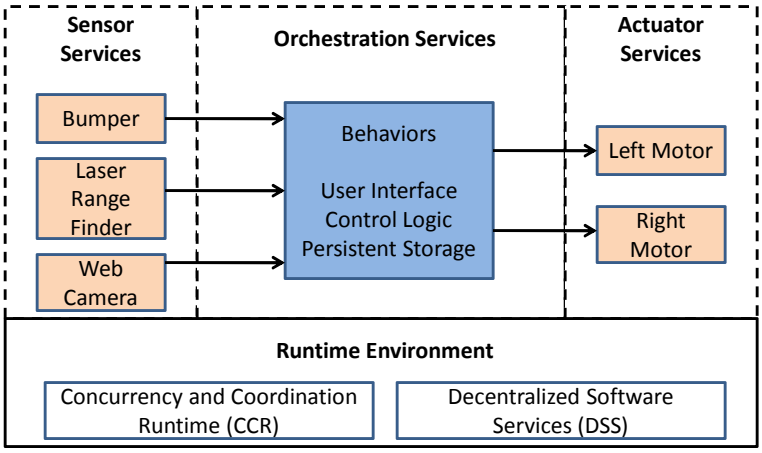
DSS Overview

- DSS adds another layer for combining CCR applications, called services, and at the same time it enables these services to run on completely separate computers and communicate via network.
- Microsoft has defined a set of *generic contracts* that describe commonly used robotics services.
- These contracts specify the APIs that must be used to communicate with robot components such as motors, sonar sensors, and even webcams. By standardizing these interfaces, the robotics community can share code more easily.

Orchestration

- Every MRDS application that you build will contain one or more services.
- Combining these services and passing messages between them, whether they are located on the same or different computers, is one of the tasks of DSS.
- The process of combining services is called *partnering*.

Orchestration – Putting Services Together



Example

- Program for Robocup Hockey

Example 1 – Creating the Project

- Demonstration of example given on page 33.
- All DSS services have a method with the name "start" and its called during the initialization of the services.
- If we code everything from scratch then create a project of type "Simple DSS Service".
- Add "using System.Threading"

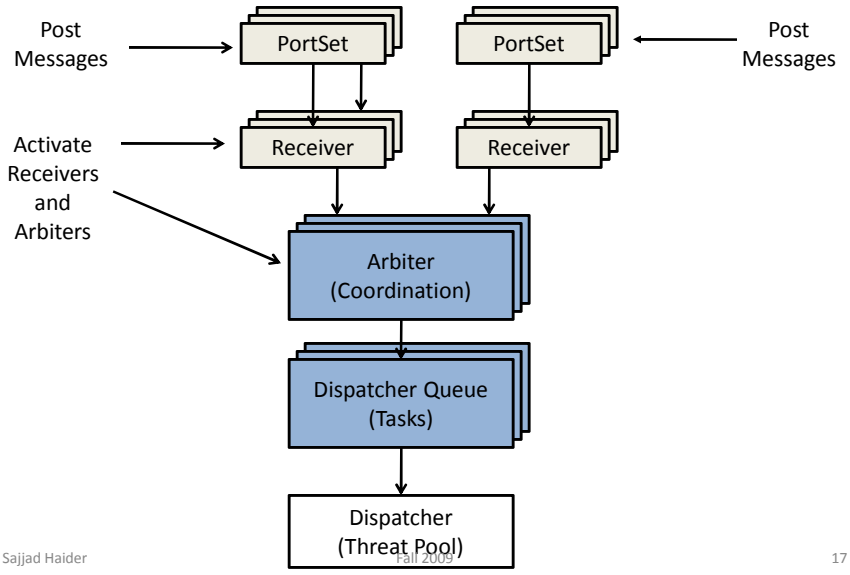
Example 1 (Cont'd)

- Each new project "Project1" has a reference to a *state* class and an *operation* class.
  - Project1
  - Main Class: Project1Services instanceof DsspServiceBase
  - State Class: Project1State
  - Operations Class: Project1Operation
- Add "using System.Threading"

Program Explanation

- *Start()* is called when the service is started by DSS.
- This is where we place code to perform any initialization that our service requires.
- To check the current thread's ID use the following command
  - Thread.CurrentThread.ManagedThreadId

CCR Architecture



CCR – Asynchronous Programming

- The CCR uses an asynchronous programming model based on *message passing* using a structure called a *port*.
- A *message* is an instance of any data type that the CLR can handle.
- **When you write DSS services, you declare your own classes for message types.**
- Ports are messages queues that only accept messages of the same data type as the one in the port declaration.

CCR - Arbiters

- In CCR terminology, you place messages into a port by *posting* to the port. **Messages remain in ports until they are dequeued by a *receiver*.**
- Activation conditions can be set on receivers to create complex logical expressions, such as a *Join* between two ports (two messages must arrive, which is effectively a logical AND) or a *Choice* between two ports (a message can arrive on either port, creating a logical OR).
- **Evaluating activation conditions is the job of *arbiters*.**

CCR – Dispatcher and Dispatcher Queues

- The coordination primitives, implemented through arbiters, are used to synchronize and control tasks.
- Once a receiver's conditions have been met, a task is queued to a *dispatcher queue* and then passed to a *dispatcher* for execution.
- Once a task is scheduled to run, a *handler* is executed. Handlers are pieces of code that run in a fully multi-threaded environment.
- Most of the time, the CCR handles all this transparently, and there is no need for you to explicitly define mutexes or write callback procedures, which you might be used to using for multi-threaded programming in the past.

DSS Provides a Hosting Environment

- The DSS is based on the CCR and does not require any other component. It provides a hosting environment for running services and making them accessible via a web browsers.
- You cannot separate DSS from the CCR.
- When you are running the examples in this chapter, you are running a DSS node and executing a service inside this node.

CCR: Key Concepts

- A key feature of the CCR is that it supports multi-threading, or *concurrency*.
- Key Concepts
  - Tasks
  - Delegates
  - Iterators

Tasks

- Dispatchers schedule tasks by allocating them to threads from a pool.
- A task contains a reference to a handler that is a piece of code you want executed.
- The DsspServiceBase class (part of DSS) provides a wrapper to execute a handler as a task called Spawn.
- The default DSS dispatcher thread pool only has two threads for a single CPU machine (even if it's a dual core). The minimum number of threads is two.

Example of a Handler

- Think about a salesman sitting in a busy pharmacist distributing tasks to different delivery boys.
- A piece of paper is a task, the medicine you need corresponds to a handler (i.e., the piece of code you want to execute)

ITask Interface

- In the CCR, tasks implement the *ITask* interface. This is useful because it gives tasks a data structure, enabling them to be passed around and queued.
- However, it also means that you must wrap a piece of code as an *ITask* before you can submit it to a dispatcher queue.

Example 1 - Spawn

```
void SimpleHandler()
{
   Console.WriteLine("Spawned Thread  " +
   Thread.CurrentThread.ManagedThreadId);

        for (int i = 0; i < 10; i++)
        {
          Wait(100);
          Console.WriteLine("Thread ID : " +
          Thread.CurrentThread.ManagedThreadId + ", " + i);
        }
}
```

Example 1 – Spawn (Cont'd)

- After *Start()*, add the following
  - Spawn(SimpleHandler);
  - Spawn(SimpleHandler);
  - Spawn(SimpleHandler);
  - Spawn(SimpleHandler);

Thread Management

- The default number of thread is 2.
- You can change it as follows
  - [ActivationSettings(ShareDispatcher = false, ExecutionUnitsPerDispatcher = 6)]