Asynchronous method invocation in Join java and polyphonic c#

Kamalakshi N
Asst Professor,
Dept of Computer Science &Engg
Saptagiri College of Engineering,
Bangalore INDIA
kamalnags@gmail.com

Dr.H.Naganna
Professor & Head
Dept of Information Science &Engg SJB Institute of Technology,Bangalore INDIA
naganna_h@hotmail.com

Abstract: In most programming languages a called method is executed synchronously, i.e. in the thread of execution from which it is invoked. If the method needs a long time to completion, e.g. because it is loading data over the internet, the calling thread is blocked until the method has finished. When this is not desired, it is possible to start a "worker thread" and invoke the method from there. In most programming environments this requires many lines of code, especially if care is taken to avoid the overhead that may be caused by creating many threads. AMI solves this problem in that it augments a potentially long-running ("synchronous") object method with an "asynchronous" variant that returns immediately, along with additional methods that make it easy to receive notification of completion, or to wait for completion at a later time. This paper focuses on asynchronous method invocation in two popular languages Join java and C#.

1 .Introduction

In (multithreaded) object-oriented programming, asynchronous method invocation (AMI), also known as asynchronous method calls or asynchronous pattern is a design pattern for asynchronous invocation of potentially long-running methods of an object.[1] It is equivalent to the IOU pattern described in 1996 by Allan Vermeulen.[2][3]. The event-based asynchronous pattern in .NET Framework and the *java.util.concurrent.FutureTask* class in Java use events to solve the same problem. This pattern is a variant of AMI whose implementation carries more overhead, but it is useful for objects representing software components.

One common use of AMI is in the active object design pattern. Alternatives are synchronous method invocation and future objects[4].An example for an application that may make use of AMI is a web browser that needs to display a web page even before all images are loaded.

The following are possible reasons for defining a method as asynchronous:

The function within the method uses the update task - The method is terminated by an event generated when the update task is successful. If you nevertheless define the method as a synchronous method under these circumstances, the workflow data may be inconsistent with the current data in the database due to update terminations or delays.

The method can also be called outside the workflow - If it is possible that a user may call the function encapsulated in the method outside the workflow although the object is also being processed under the control of a workflow at the same time, you should implement the method as an asynchronous method. It is therefore irrelevant whether calling the function takes place inside or outside the workflow system. The workflow continues after the event occurs.

An asynchronous method with comprehensive functionality is called in the task - This task gets its specific character from the terminating events. e.g., In a task, a specific order is to be released for billing. In the task, call the (general) method Process order and enter the event Billing block deleted as the terminating event.

It is also to be possible for the task to be terminated by events not necessarily from method processing - The task in which the method Process sales order is executed is also terminated by the event Customer unable to pay

2Join Java

Join Java is a programming language that extends the standard Java programming language with the join semantics of the join-calculus. It was written at the University of South Australia within the Reconfigurable Computing Lab by Dr. Von Itzstein.

The Join Java extension introduces three new language constructs:

- Join methods
- Asynchronous methods
- Order class modifiers for determining the order that patterns are matched

Concurrency in most popular programming languages is implemented using constructs such as semaphores and monitors. Libraries are emerging (such as the Java concurrency library JSR-166) that provide higher-level concurrency semantics. Communicating Sequential Processes (CSP), Calculus of Communicating Systems (CCS) and Pi have higher-level synchronization behaviours defined implicitly through the composition of events at the interfaces of concurrent processes. Join calculus, in contrast, has explicit synchronization based on a localized conjunction of events defined as reduction rules. Join semantics try to provide explicit expressions of synchronization without breaching the object-oriented idea of modularization, including dynamic creation and destruction of processes and channels.

The Join Java language can express virtually all published concurrency patterns without explicit recourse to low-level monitor calls. In general, Join Java programs are more concise than their Java equivalents. The overhead introduced in Join Java by the higher-level expressions derived from the Join calculus is manageable. The synchronization expressions associated with monitors (wait and notify) which are normally located in the body of methods can be replaced by Join Java expressions (the Join methods) which form part of the method signature.

Join methods

A Join method is defined by two or more Join fragments. A Join method will execute once all the fragments of the Join pattern have been called. If the return type is a standard Java type then the leading fragment will block the caller until the Join pattern is complete and the method has executed. If the return type is of type **signal** then the leading fragment will return immediately. All trailing fragments are asynchronous so will not block the caller.

Example:

```
class JoinExample {
   int       fragment1()       &
fragment2(int x) {
      //will return value of x
      //to caller of fragment1
      return x;
   }
}
```

Ordering modifiers

Join fragments can be repeated in multiple Join patterns so there can be a case when multiple Join patterns are completed when a fragment is called. Such a case could occur in the example below if B(), C() and D() then A() are called. The final A() fragment completes three of the patterns so there are three possible methods that may be called. The **ordered** class modifier is used here to determine which Join method will be called. The default and when using the **unordered** class modifier is to pick one of the methods at random. With the **ordered** modifier the methods are prioritised according to the order they are declared.

Example:

```
class  ordered  SimpleJoinPattern
{
   void A() & B() {
   }
   void A() & C() {
   }
   void A() & D() {
   }
   signal D() & E() {
   }
}
```

Asynchronous methods

Asynchronous methods are defined by using the **signal** return type. This has the same characteristics as the **void** type except that the method will return immediately. When an asynchronous method is called a new thread is created to execute the body of the method.

Example:

```
class ThreadExample {
   signal thread(SomeObject x) {
      //this  code  will  execute
in a new thread
   }
}
```

C Sharp is a multi-paradigm programming language that supports imperative, generic and object-oriented programming. It is a part of the Microsoft .NET Framework. It is similar to C++ in its object-oriented syntax and is also influenced by Java and Delphi. Polyphonic C# extends C#. MC# is an extension of Polyphonic C# that can work on the .NET platform. C-omega is an extension to C# that succeeded Polyphonic C#. It enables access to data stores and includes constructs that support concurrent programming.

3.Polyphonic C#

Polyphonic C# adds just two new concepts: *asynchronous methods* and *chords*.

Asynchronous Methods. Conventional methods are synchronous, in the sense that the caller makes no progress until the callee completes. In Polyphonic C*]*, if a method is declared *asynchronous* then any call to it is guaranteed to complete essentially immediately. Asynchronous methods never return a result (or throw an exception); they are declared by using the async keyword instead of void. Calling an asynchronous method is much like sending a message, or posting an event.Since asynchronous methods have to return immediately, the behaviour of a method such as async *postEvent*(*EventInfo data*) *f* // large method body *g* is the only thing it could reasonably be: the call returns immediately and 'large method body' is scheduled for execution in a different thread (either a new one spawned to service this call, or a worker from some pool). However, this kind of definition is actually rather rare in Polyphonic C*]*. More commonly, asynchronous methods are defined using chords, as described below, and do not necessarily require new threads.

Chords. A *chord* (also called a 'synchronization pattern', or 'join pattern') consists of a header and a body. The header is a set of method declarations separated by '&'. The body is only executed once *all* the methods in the header have been called. Method calls are implicitly queued up until/unless there is a matching chord.
Consider for example
public class *Buffer {*
public string *Get*() & public async *Put*(string *s*) {
return *s*;
}
}
The code above defines a class *Buffer* with two instance methods, which are jointly defined in a single chord. Method string *Get*() is a synchronous method taking no arguments and returning a string. Method async *Put*(string *s*) is asynchronous (so returns no result) and takes a string argument.
If *buff* is a instance of *Buffer* and one calls the synchronous method *buff .Get*() then there are two possibilities:
—If there has previously been an unmatched call to *buff .Put*(*s*) (for some string *s*) then there is now a match, so the pending *Put*(*s*) is dequeued and the body of the chord runs, returning *s* to the caller of *buff .Get*().
—If there are no previous unmatched calls to *buff .Put*(.) then the call to *buff .Get*() blocks until another thread supplies a matching *Put*(.).
Conversely, on a call to the asynchronous method *buff .Put*(*s*), the caller never waits, but there are two possible behaviours with regard to other threads:
—If there has previously been an unmatched call to *buff .Get*() then there is now a match, so the pending call is dequeued and its associated blocked thread is
awakened to run the body of the chord, which returns *s*.
—If there are no pending calls to *buff .Get*() then the call to *buff .Put*(*s*) is simply queued up until one arrives.
Exactly *which* pairs of calls are matched up is unspecified, so even a single-threaded program such as
Buffer buff = new *Buffer*();
buff .Put(”blue”);

```
buff .Put(”sky”);
Console.Write(buff .Get() + buff .Get());
```

is non-deterministic (printing either ”bluesky” or ”skyblue”).

Note that the implementation of *Buffer* does not involve spawning any threads: whenever the body of the chord runs, it does so in a preexisting thread (viz. the one that called *Get*()). The reader may at this point wonder what are the rules for deciding in which thread a body runs, or how we know to which method call the final value computed by the body will be returned. The answer is that in any given chord, at most one method may be synchronous. If there is such a method, then the body runs in the thread associated with a call to that method, and the value is returned to that call. Only if there is no such method (i.e. all the methods in the chord are asynchronous) does the body run in a new thread, and in that case there is no value to be returned.

It should also be pointed out that the *Buffer* code, trivial though it is, is threadsafe.The locking that is required (for example to prevent the argument to a single *Put* being returned to two distinct *Get*s) is generated automatically by the compiler.More precisely, deciding whether any chord is enabled by a call and, if so, removing the other pending calls from the queues and scheduling the body for execution is an atomic operation. Apart from this atomicity guarantee, however, there is *no*monitor-like mutual exclusion between chord bodies. Any mutual exclusion that is required must be programmed explicitly in terms of synchronization conditions in chord headers.

The *Buffer* example uses a single chord to define two methods. It is also possible (and common) to have multiple chords involving a given method. For example:

```
public class Buffer {
public string Get() & public async Put(string s) {
return s;
}
public string Get() & public async Put(int n) {
return n.ToString();
}

}
```

The second new syntax that is enabled in chords is the ability to “join” methods with each other. Two or more methods joined with each other are what we refer to as a “chord.” All method calls are kept in a queue, and the body of a chord is executed when all the joined methods in a chord are present in the queue. The syntax for joining a chord is simply listing all the methods in the chord separated by an ampersand (“&”). Going back to the previous example,

```
public class ClassUsingAsyncVersion2
{
public async m1();
public async m2();
when m1() & m2()
{
//method stuff
}
}
```

As is the case here, chords may be composed entirely of asynchronous methods. In this case, each call to m1() or m2() completes instantaneously as normal and the calling thread continues immediately following the call. Each object has an underlying queue in which it stores all calls to methods that have not been part of a completed chord. When all the methods of a chord are found in this queue, and the chord is composed of only asynchronous methods as in the example above, the chord body is not executed in any of the calling threads. After all, how would we decide which one to interrupt? Instead, a new thread is created, or an available thread is drawn from a thread pool, depending on the implementation.

It is also possible for a chord to return a value if one of the methods in the chord is synchronous. That is, only one method may have a return type other than "async." When a thread makes a call to the synchronous method in a chord, it blocks until the chord is completed, at which point the body of the chord is executed in the same thread of the synchronous method. This is important to note: a chord that contains a synchronous method does not result in the creation of a new thread. Building on the previous example, here is a class with a synchronous chord:

```
public class ClassUsingAsyncVersion3
{
public async m1();
public async m2();
when m1() & m2()
{
//method stuff
}
public int m3() & m1()
{
return 0;
}
}
```

If a thread makes a call to m3() without any previous calls to m1() by any other threads, it
will block and wait to be notified. It will be notified when another thread makes a call to m1() and the chord is complete, and the thread that called m3() will then execute the code
in the body of the chord, in this case simply returning "0." There is one final important rule to consider about the behavior of chords. We must decide what to do when there is more than one possible chord that could be executed in the queue, but the two chords share a method. For example, in the third version of the ClassWithAsync, as seen above, m1() is part of two different chords. In the program above, it might be the case that there are calls to m2() and m3() already made, and a call to m1() completes two chords at the same time. We must decide which chord executes and which must wait for another call to the shared method. In the original Polyphonic C# paper, it is stated that generally this decision will be nondeterministic, i.e. one of them will be chosen at random, and you should not write your program to rely on a specific behavior. A major subject of this paper is an alternative to this non-determinism that might make chords more useful.

4.CONCLUSION

This paper discusses on the overview Join Java and Polyphonic C#

REFERENCES

[1] "Asynchronous Method Invocation". *Distributed Programming with Ice*. ZeroC, Inc.. http://www.zeroc.com/doc/Ice-3.2.1/manual/Async.34.2.html#71139. Retrieved 22 November 2008.

[2] Vermeulen, Allan (June 1996). "An Asynchronous Design Pattern". *Dr. Dobb's Journal*. http://www.ddj.com/184409898. Retrieved 22 November 2008.

[3] Nash, Trey (2007). "Threading in C#". *Accelerated C# 2008*. Apress. ISBN 9781590598733.

[4] Lavender, R. Greg; Douglas C. Schmidt (PDF). "*Active Object*:" http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf. Retrieved 22 November 2008.

[5] Nick benton, Luca cardelli, and Ce´dric fournet "Modern

Concurrency Abstractions for C#” Microsoft Research Lab 2004

[6] Joel Barciauskas “Extensions to Polyphonic C# “ 2006