Jingle Bells: Solving the Santa Claus Problem in Polyphonic C$^\sharp$

Nick Benton
Microsoft Research
Cambridge UK
nick@microsoft.com

March 20, 2003

Abstract

The Santa Claus problem is an interesting exercise in concurrent programming which has been used in a comparison of the concurrency mechanisms of Ada and Java. We present a simple solution to the problem in Polyphonic C$^\sharp$, an extension of C$^\sharp$ with new concurrency constructs based on the Join calculus.

1 Introduction

1.1 The Problem

Originally due to Trono [6], the Santa Claus problem is an interesting (and amusing) exercise in concurrent programming which is a little more challenging than traditional mutual-exclusion problems, as it involves three sorts of process and also requires a number of processes to cooperate. The problem may be stated as follows:

> Santa repeatedly sleeps until wakened by either all of his nine reindeer, back from their holidays, or by a group of three of his ten elves. If awakened by the reindeer, he harnesses each of them to his sleigh, delivers toys with them and finally unharnesses them (allowing them to go off on holiday). If awakened by a group of elves, he shows each of the group into his study, consults with them on toy R&D and finally shows them each out (allowing them to go back to work).
>
> Santa should give priority to the reindeer in the case that there is both a group of elves and a group of reindeer waiting.

The errors which are easy to make in an attempt to solve this problem include extra elves being able to sneak into a group once Santa has started showing elves into his office, or Santa being able to go off delivering toys whilst the reindeer are still waiting in the stable. Ben-Ari [1] points out a bug (of the second kind) in Trono's initial semaphore-based solution, shows how the problem may be solved fairly neatly using Ada's concurrency primitives and compares this with a much less elegant and less efficient solution in Java.

1.2 The Language

C$^\sharp$ [3] is a modern, type-safe, object-oriented programming language which runs on Microsoft's .NET platform. Its concurrency model is similar to that of Java, with a re-entrant lock associated with each heap allocated object. A thread executing a `lock` statement aquires the lock on a specified object during the execution of a block of code. The library calls `Wait`, `Pulse` and `PulseAll` behave like `wait`, `notify` and `notifyAll` in Java. A variety of other traditional shared-memory concurrency primitives, inherited from the Win32 API, are available in the .NET class libraries.

Polyphonic C$^\sharp$ [2] is an extension of C$^\sharp$ with new concurrency constructs, based on the Join calculus [5, 4]. Methods in Polyphonic C$^\sharp$ may be defined as either synchronous, in which case the caller blocks until the method completes (as is usually the case for method calls), or asynchronous, in which case the caller never blocks and there is no return value. The more novel feature of the language is *chords*, which associate a code body with a *set* of method headers. The body of a chord can run only once all the methods in the set have been called. Calls to synchronous methods block until/unless there is a matching chord, whilst the arguments of calls to asynchronous methods are internally queued until/unless they can be consumed by a matching chord. A particular method may appear in more than one chord and in the case that multiple chords match it is unspecified which will execute. If a chord involving *only* asynchronous methods matches, then its body runs in a new thread. The body of a chord containing a synchronous method will run in (and return a value to) the thread which called that method – there can be at most one synchronous method in a chord.

Polyphonic C$^\sharp$ is intended as a language for orchestrating distributed applications built on asynchronous messaging. However, its model of concurrency and synchronization is equally applicable to programming with multiple threads in a shared-memory environment. Concurrency abstractions such as semaphores, reader-writer locks, barriers and active objects, as well as more application-specific mechanisms such as custom thread pools, can easily be implemented in Polyphonic C$^\sharp$.

2 The Solution

2.1 An auxiliary class

We begin by defining a class `nway` for allowing one thread to synchronize with a number of others. One thread can call `acceptn(`m`)` on an `nway` and will then block until m other threads have each called `entry()`. Conversely, calls to `entry()` will block until there has been a matching call to `acceptn(`m`)`.[1]

```
public class nway {
   public void entry() & private async tokens(int n) {
      if (n==1) allgone();
      else tokens(n-1);
```

[1]Each call to `entry()` unblocks immediately, rather than waiting for $m-1$ other calls to have been made, which corresponds to the way in which rendezvous are used in Ben-Ari's solution.

```
   }

   public void acceptn(int n) {
      tokens(n);
      wait();
   }

   private void wait() & private async allgone() {
   }
}
```

The definition of `nway` illustrates a common pattern in Polyphonic C$^\sharp$ – the use of private messages to carry state. A call to `acceptn(`m`)` generates a private message `tokens(`m`)`. Each call to `entry()` waits for and then consumes a `tokens(`k`)` message and generates a new message of the form `tokens(`$k-1$`)` or `allgone()`. The `allgone()` message is then consumed by, and unblocks, the call to `wait()`.

The definition of `nway` is equivalent to one built on a binary rendezvous, which is closer to the pattern used in Ada. The code for the alternative version is shown in Appendix A.

2.2 Basic solution

We will use instances of `nway` to synchronize Santa with all the reindeer during harnessing and unharnessing, and with all the elves in a group whilst he is showing them in and showing them out of his office:

```
static nway harness = new nway();
static nway unharness = new nway();
static nway roomin = new nway();
static nway roomout = new nway();
```

Now each elf thread runs the following code:

```
while (true) {
   Work();             // until finding a problem
   elfqueue();         // try to join group of 3
   roomin.entry();     // wait for Santa to show me in
   ConsultWithSanta(); // until problem solved
   roomout.entry();    // wait for Santa to show me out
}
```

The call to `elfqueue()` is used to control the marshalling of elves into groups of three and to wake Santa up. It synchronizes with another private message counting the number of elves waiting:

```
static void elfqueue() & static async elveswaiting(int e) {
   if (e==2) elvesready();    // last elf in a group
   else elveswaiting(e+1);
}
```

Initially, there is a message `elveswaiting(0)`. Each elf calling `elfqueue()` will wait to consume an `elveswaiting(`k`)` message and then either send `elveswaiting(`$k+$

1) or `elvesready()`, which will signal Santa to wake up. Observe that in the latter case, an `elveswaiting()` message is not sent – any further elves calling `elfqueue()` will be blocked until this group have been shown in.

The code for the reindeer is essentially the same as that for the elves. One message `reinwaiting(`k`)` counts how many reindeer have returned from holiday, whilst `reindeerready()` is used to signal Santa that they are all back:

```
  // each reindeer thread runs this
  while (true) {
    Holiday();         // until sufficiently refreshed
    reindeerback();    // join group waiting in stable
    harness.entry();   // wait for Santa to harness me up
    DeliverToys();     // until finished
    unharness.entry(); // wait for Santa to unharness me
  }

 static void reindeerback() & static async reinwaiting(int r) {
   if (r==8) reindeerready();      // last reindeer
   else reinwaiting(r+1);
 }
```

Santa himself simply runs

```
  while (true) {
     waittobewoken();
  }
```

where the synchronous method `waittobewoken()` is defined in two chords, one of which synchronizes with `reindeerready()` and one with `elvesready()`:

```
 static void waittobewoken() & static async reindeerready() {
    harness.acceptn(9);   // harness all nine
    reinwaiting(0);       // reset waiting count
    DeliverToys();        // with the reindeer
    unharness.acceptn(9); // unharness them all
 }

 static void waittobewoken() & static async elvesready() {
    roomin.acceptn(3);    // show three elves into office
    elveswaiting(0);      // now allow others to join waiting group
    ConsultWithElves();   // until finished
    roomout.acceptn(3);   // show them all out
 }
```

Each of the chord bodies does the appropriate thing, including resetting the count of waiting elves or reindeer. The placing of `reinwaiting(0)` is not particularly critical – it could be placed anywhere within the chord or even along with `reindeerready()` in the `reindeerback()` chord. On the other hand, to prevent queue-jumping, `elveswaiting(0)` should not be signalled until all three elves in the group have been shown into the office. Placing it where it is in the code above allows a new group of three to assemble whilst Santa is dealing with an earlier group.

2.3 Priorities

Under our current implementation of Polyphonic C$^\sharp$, the solution in the previous section *does* give priority to the reindeer, as required in the original problem statement. This is because the check for a matching chord follows the textual ordering of the original source code and the chord matching `waitobewoken()` with `reindeerready()` precedes that which matches it with `elvesready()`. Thus even if it was an `elvesready()` message which was originally responsible for scheduling the Santa thread for execution, if there is a `reindeerready()` message present when the Santa thread actually starts executing then it will be the reindeer chord which gets executed. This behaviour is observed in practice, as is the starvation of the reindeer by the elves in the case that the clauses are reversed.[2]

Relying on the matching order is highly unsatisfactory: this is intentionally not part of the official semantics of Polyphonic C$^\sharp$ and a version of the compiler which attempted to do more optimization would almost certainly not preserve the original textual order. Moreover, this is a clumsy and error-prone way in which to control priorities and in more complex cases there might be no textual ordering which expressed the desired priorities.

Fortunately, programming the priorities explicitly is easy. We introduce a single new asynchronous message `reindeernotready()` and add this to the synchronization between Santa and a ready group of elves:

```
static void waittobewoken() & static async elvesready()
& static async reindeernotready() { // new line
   reindeernotready();             // new line
   // rest of body as before
}
```

We then consume the `reindeernotready()` message when the reindeer *are* ready:

```
static void reindeerback() & static async reinwaiting(int r) {
  if (r==8) {
     clearreindeernotready(); // new line
     reindeerready();
  }
  else reinwaiting(r+1);
}

// new chord
static void clearreindeernotready() & static async reindeernotready() {}
```

and add a send of `reindeernotready()` to the initialization code and to the Santa/reindeer chord, which now looks like this:

[2]On my Windows XP machine with all threads at the same priority and the same uniformly distributed 0 to 3 second delay for Santa's consultations and toy deliveries, each elf working and each reindeer holidaying, Santa serves about 2.3 times as many groups of 3 elves as he does groups of 9 reindeer. When the order of the two chords is reversed, he serves about 21 times as many elf groups as reindeer groups. (Of course, the reindeer are unlikely to be literally starving – they are stuck in the stable with some hay instead of flying around in the snow. . .)

```
static void waittobewoken() & static async reindeerready() {
   harness.acceptn(9);
   reindeernotready();  // new line
   reinwaiting(0);
   DeliverToys();
   unharness.acceptn(9);
}
```

This simple six-line addition is all we need to control the priorities explicitly, and does indeed show the correct behaviour in practice.[3]

2.4 Conclusion

This solution to the Santa Claus problem is shorter and, I believe, simpler, more elegant and easier to reason about than Ben-Ari's solution using Ada's more complex synchronization mechanisms. It also seems likely to be slightly more efficient, since an `nway` does fewer context switches than using multiple binary rendezvous.

Our solution is undeniably an improvement in every way over his attempt to solve the problem in Java (and the same problems would arise if one were to attempt a solution in unmodified C$^\sharp$).

This provides another small piece of evidence that Join calculus-based concurrency primitives, as well as being a good basis for distributed computation, are a very attractive choice for solving traditional concurrency problems.

References

[1] M. Ben-Ari. How to solve the santa claus problem. *Concurrency: Practice & Experience*, 10(6):485–496, 1998.

[2] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C$^\sharp$. In *Proceedings of ECOOP 2002*, volume 2374 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[3] ECMA. Standard ECMA-334: C$^\sharp$ Language Specification, December 2001.

[4] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, Ecole Polytechnique, Palaiseau, November 1998. INRIA TU-0556. Also available from `http://research.microsoft.com/~fournet`.

[5] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM-SIGACT Symposium on Principles of Programming Languages*, pages 372–385. ACM, January 1996.

[6] J. A. Trono. A new exercise in concurrency. *SIGCSE Bulletin*, 26(3):8–10, 1994. Corrigendum: 26(4):63.

[3] With everything else as before, this version gives an elf group to reindeer group service ratio of 2.42 when the reindeer chord is first and 2.45 when the elf chord is first. The difference is due to a small race between the last elf and the last reindeer to grab the `reindeernotready()` message.

A N-way synchronization using 2-way

Here we give an alternative definition for `nway` which corresponds more closely to the pattern used in Ada. We first define a binary rendevous class `twoway` for synchronizing two threads. One thread calls `entry()`, the other calls accept() and whichever arrives first is blocked until the other has also arrived. An `nway` then captures the pattern of one thread calling `accept()` n times to synchronize with n other thread, each of which calls `entry()` once:

```
public class twoway {
   public void entry() {
      gotentry();
      waitforaccept();
   }

   public void accept() {
      gotaccept();
      waitforentry();
   }

   private void waitforentry() & private async gotentry() {
   }

   private void waitforaccept() & private async gotaccept() {
   }
}

public class nway {
   private twoway rv = new rv();

   public void entry() {
      rv.entry();
   }

   public void acceptn(int n) {
      for(int i=0;i<n;i++)
         rv.accept();
   }
}
```

This alternative is less efficient than our original definition, however, since the `accept`ing process can be repeatedly scheduled and blocked in the loop.