Concurrent Programming

Gavin Lowe and Bernard Sufrin

Hilary Term 2016-17

7: Interacting Peers

Reading: Andrews, Section 7.4.

Interacting peers

A common paradigm of interaction in concurrent programming is *interacting peers*. Several processes execute basically the same code, and exchange messages to achieve some task.

In this chapter we will examine four patterns of interacting peers:

* A system with a centralized process doing most of the work;
* A fully-connected topology, where each process sends messages to all the others;
* A ring topology, where each process communicates with just its two neighbours;
* A binary tree topology, where each process communicates just with its parent and two children.

We will illustrate the patterns with a simple example: at the start, each node holds an integer; at the end, each node should hold the minimum and maximum of all those integers.

The happens-before relation, $\preceq$

It can be useful to consider an ordering over actions (memory reads and writes, and sends and receives of messages) that talks about the order in which those actions must happen.

We will write $a \preceq a'$, and say a happens before a', if in all executions of the program, a' can occur only after, or simultaneously with, a.

Note that if a and a' are a send and the corresponding receive over a synchronous channel, then $a \preceq a'$ and $a' \preceq a$, so the relation is not antisymmetric.

Formally, $\preceq$ is a preorder, i.e. a transitive, reflexive order such that:

* If a and a' are actions of a single thread, and a precedes a', then $a \preceq a'$;
* If a is a send and a' is the corresponding receive, then $a \preceq a'$ (regardless of whether the communication is over a synchronous or asynchronous channel);
* If a is a receive and a' is the corresponding send, and the communication is over a synchronous channel, then $a \preceq a'$.

Note that some pairs of actions may be unrelated by $\preceq$, so they can happen in either order (necessarily in different threads). In such cases, we need to be sure that either order is acceptable — i.e. there is no race condition.

Reasoning about patterns

We will reason about the patterns by identifying *invariants* that hold at certain points in the execution.

We will also consider the *total* number of messages sent.

Finally, we will consider the number of messages sent *sequentially*. Recall the "happens-before" relation $\preceq$. We will say that messages m_1, ..., m_n form a $totally\text{-}ordered\ chain$ if

$$m_1 \prec m_2 \prec \ldots \prec m_n.$$

We will be interested in identifying the maximum length chain.

Centralized pattern

In this protocol, each client node sends its value to a central node, which calculates the minimum and maximum, and sends those back.

We use a single channel in each direction, which is shared by the client processes:

```
type IntPair = (Int, Int)
val toController = ManyOne[Int]
val fromController = OneMany[IntPair]
```

The clients

```
def Client(me: Int, in: ?[IntPair], out: ![Int]) = proc
{
  // Choose value randomly
  val v = random·nextInt(1000)
  println("Client␣"+me+"␣chooses␣value␣"+v)

  // Send value to controller
  out!v

  // Receive min and max from controller
  val (min,max) = in?;
  println("Client␣"+me+"␣ends␣with␣values␣"+(min,max))
}
```

The controller

```
def Controller(in: ?[Int], out: ![IntPair]) = proc
{
  // Choose value randomly
  val v = random.nextInt(1000)
  println("Controller chooses value "+v)

  // Receive values, and calculate min and max
  var min = v; var max = v;
  for (i ← 1 until N) {
    val w = in?; if(w<min) min=w; if(w>max) max=w;
  }
  println("Controller ends with values "+(min,max));

  // Distribute min and max
  for (i ← 1 until N) out!(min,max);
}
```

Invariant

Write v_i for the value chosen by node i. Then during the first stage of the protocol, the following invariant holds:

$$\texttt{min} = \min(\{v_0\} \cup \{v_i \mid 1 \leqslant i < N \wedge \text{node } i \text{ has sent its value}\}),$$
$$\texttt{max} = \max(\{v_0\} \cup \{v_i \mid 1 \leqslant i < N \wedge \text{node } i \text{ has sent its value}\}).$$

Efficiency

This protocol uses $2(N-1)$ messages in total. However, none of these messages can occur concurrently, since all messages involve the controller. We can find a totally-ordered chain of communications

$$\texttt{toController}.i_1 \prec \texttt{toController}.i_2 \prec \ldots \prec \texttt{toController}.i_{N-1} \prec \texttt{fromController}.j_1 \prec \texttt{fromController}.j_2 \prec \ldots \texttt{fromController}.j_{N-1}$$

for some permutations $i_1, \ldots, i_{N-1}$ and $j_1, \ldots, j_{N-1}$ of $1, \ldots, N-1$.

Symmetric pattern

In this solution, every node sends its value to every other node, which calculates the minimum and maximum.

We give each node its own channel on which it can receive messages.

```
val toNode = for (i←0 until N) yield ManyOne[Int];
```

A node

The node needs to send $N-1$ messages, and receive $N-1$ messages. It is easiest if we do these two tasks in parallel.

```
def Node(me: Int, in: ?[Int], toNode: Seq[![Int]]) = proc
{
  // Choose value randomly
  val v = random.nextInt(1000)
  println("Node "+me+" chooses value "+v)

  // Process to distribute value to all other nodes
  def Sender = proc
  {
    for (i ← 0 until N) { if(i!=me) toNode(i)!v  }
  }

  ...
}
```

A node

```
def Node(me: Int, in: ?[Int], toNode: Seq[![Int]]) = proc
{
   ...
   // Process to receive values from other nodes,
   // and calculate min and max
   def Receiver = proc
   { var min=v; var max=v;
     for (i ← 1 until N)
     {
       val w = in?; if (w<min) min=w; if(w>max) max=w;
     }
     println("Node␣"+me+"␣ends␣with␣values␣"+(min,max))
   }

   // Run sender and receiver in parallel
   (Sender || Receiver)()
}
```

Invariant

Write v_j for the value chosen by node j. Then each node i has values for `min` and `max` such that:

$$\texttt{min} = \min(\{v_i\} \cup \{v_j \mid \text{node } j \text{ has sent its value to node } i\}),$$
$$\texttt{max} = \max(\{v_i\} \cup \{v_j \mid \text{node } j \text{ has sent its value to node } i\}).$$

Removing contention

Each `Sender` sends to the other nodes in order, starting from `0`. This means that:

* All the nodes are contending to send to node `0`, so most will be temporarily blocked;
* Node `N-1` has to wait until another node has finished sending to all other nodes before receiving *anything*. This gives a chain of length $2N - 3$.

A better way is for each `Sender` to send in a different order, say starting from the one with identity one higher than itself:

```
def Sender = proc
{
  for (i ← 1 until N) { toNode((me+i)%N)!v }
}
```

Now the longest chain is of length $N - 1$.

Efficiency

This protocol uses $N(N-1)$ messages. However, they can be sent in just $N-1$ communication rounds (once contention has been removed).

A ring topology

We will see two protocols using a (logical) ring, where each node i sends messages to node $(i + 1) \bmod N$ and receives from node $(i - 1) \bmod N$.

In the first protocol, a single token is passed twice round the ring.

In the second protocol, N tokens are simultaneously passed round the ring.

The first ring topology protocol

The first protocol uses two stages. During the first stage, node `0` acts as the initiator, by sending a token containing two copies of its value.

Each node in turn receives the token, containing the minimum and maximum value so far, calculates the new minimum and maximum, taking its own value into account, and sends them on.

More precisely, the values `(min1, max1)` passed from node i to node $(i+1) \bmod N$ will satisfy:

$$\texttt{min1} = \min(\{v_j \mid 0 \leqslant j \leqslant i\}), \qquad \texttt{max1} = \max(\{v_j \mid 0 \leqslant j \leqslant i\}).$$

When the values gets back to node `0` they equal the overall minimum and maximum, $\min(\{v_j \mid 0 \leqslant j \leqslant N-1\})$ and $\max(\{v_j \mid 0 \leqslant j \leqslant N-1\})$. These values are then passed around the ring in the second stage, and each node records the values.

A normal node

```
def Node(me: Int, in: ?[IntPair], out: ![IntPair]) = proc
{
  // Choose value randomly
  val v = random·nextInt(1000)
  println("Node␣"+me+"␣chooses␣value␣"+v)

  // receive min and max so far, and pass on
  // possibly updated min and max
  val (min1, max1) = in?;
  out!(Math·min(min1,v), Math·max(max1,v))

  // receive final min and max
  val (min, max) = in?;
  out!(min,max)
  println("Node␣"+me+"␣ends␣with␣values␣"+(min,max))
}
```

The initiator

```
def Initiator(in: ?[IntPair], out: ![IntPair]) = proc
{
  // Choose value randomly
  val v = random.nextInt(1000)
  println("Initiator chooses value "+v)

  // Start the communications going
  out!(v,v)

  // Receive min and max back, and send them round
  val (min,max) = in?;
  out!(min,max)

  // Receive them back at the end
  in?;
  println("Initiator ends with values "+(min,max))
}
```

Efficiency

The previous protocol used $2N$ messages, sent sequentially. Each node is inactive for most of the time. This pattern is most effective for problems where each node can do computation between sending a message and receiving the next.

The second ring protocol uses more messages —a total of N^2— but only N rounds. Each node is active for most of the time (so one slow node will slow down the whole ring).

A second ring protocol

Each value gets passed around the ring. Each node keeps track of the minimum and maximum values it has seen so far.

More precisely, on round k, each node i receives a value from node $(i-1) \bmod N$ that originated with node $(i-k) \bmod N$; it sends this value on to node $(i+1) \bmod N$, and keeps track of the minimum and maximum values, `(min, max)` it has seen so far:

$$\begin{aligned} \texttt{min} &= \min(\{v_{(i-j) \bmod N} \mid 0 \leqslant j \leqslant k\}), \\ \texttt{max} &= \max(\{v_{(i-j) \bmod N} \mid 0 \leqslant j \leqslant k\}). \end{aligned}$$

At the end of round $N-1$, it holds the overall minimum and maximum values.

A node

```
def Node(me: Int, in: ?[Int], out: ![Int]) = proc
{
  val v = random.nextInt(1000)
  println("Node "+me+" chooses value "+v)
  out!v

  // Repeatedly receive value from neighbour,
  // update max and min, and pass value on
  var min = v; var max = v;
  for (k ← 1 until N) {
    val w = in?;
    if (w<min) min = w; if (w>max) max = w;
    out!w
  }
  val w = in?; assert(v==w)
  println("Node "+me+" ends with values "+(min,max))
}
```

Buffering

This protocol needs some buffering between nodes. Why?

Buffers of size one are enough to ensure correctness. However, a bit more buffering might help to overcome inconsistencies in speeds of nodes.

Tree-based protocol

The final protocol works by arranging nodes into a binary tree. In the initial stage, values get passed up the tree, starting from the leaves: each node passes to its parent the minimum and maximum of the subtree for which it is the root. At the end of this stage, the root of the tree obtains the overall minimum and maximum. In the second stage, these values get passed back down the tree.

Tree-based protocol

More precisely, we will arrange the nodes into a binary *heap* (as in heap sort). Node 0 is the root of the tree. Node i is the parent of nodes $2i+1$ and $2i+2$ (if those nodes exist).

We will use two sequences of channels, to pass data up and down the tree:

```
val up   = for (i←0 until N) yield OneOne[IntPair](s"up$i")
val down = for (i←0 until N) yield OneOne[IntPair](s"down$i")
```

The sequences are indexed by the *child* node: `up(i)` and `down(i)` are used to communicate between node i and its parent, node $(i-1)$ div 2.

A node

```
def Node(me: Int) = proc("Node"+me)
{
  // Choose value randomly
  val v = random·nextInt(1000)
  println("Client␣"+me+"␣chooses␣value␣"+v)
  var min = v; var max = v

  val child1 = 2*me+1; val child2 = 2*me+2
  // Receive min and max values from both children
  if (child1<N){
    val (min1, max1) = up(child1)?;
    if(min1<min) min=min1; if(max1>max) max=max1;
    }
  if(child2<N){
    val (min2, max2) = up(child2)?;
    if(min2<min) min=min2; if(max2>max) max=max2;
  }
```

```
def Node(me: Int) = proc("Node"+me)
{
  ...
  // Send min and max to parent,
  // and wait for overall min and max to return
  if (me!=0) {
    up(me)!(min,max)
    val (_min, _max) = down(me)?; min = _min; max = _max
  }

  // Send min and max to children
  if(child1<N) down(child1)!(min,max);
  if(child2<N) down(child2)!(min,max);

  println("Node␣"+me+"␣ends␣with␣values␣"+(min,max));
}
```

Correctness

Claim: Each node i (with $0 < i < N$) passes to its parent the minimum and maximum of the values held by the nodes in the subtree rooted at i:

$$(\min\{v_j \mid j \in descendants(i)\}, \max\{v_j \mid j \in descendants(i)\})$$

where

$$descendants(i) = \{i\} \cup (\text{if } 2i+1 < N \text{ then } descendants(2i+1) \text{ else } \{\}) \\ \cup (\text{if } 2i+2 < N \text{ then } descendants(2i+2) \text{ else } \{\}).$$

The claim can be proven by induction on the size of $descendants(i)$.

Hence node 0 ends up with the overall minimum and maximum, which get passed back down the tree.

Efficiency

This protocol uses $2(N-1)$ messages (each node except 0 sends one message on up, and receives one message on down). It uses about $4\lfloor \log N \rfloor$ rounds.

Comparison

Algorithm	messages	rounds
Centralised	$2(N-1)$	$2(N-1)$
Symmetric	$N(N-1)$	$N-1$
Ring (1)	$2N$	$2N$
Ring (2)	N^2	N
Tree	$2(N-1)$	$\sim 4\lfloor \log N \rfloor$

Summary

* Interacting peers;
* Different patterns of interaction: centralised, symmetric, ring, tree;
* Reasoning about distributed algorithms;
* Efficiency: total number of messages, and number of sequential messages (rounds).

Contents