Project Report

Kamen Yotov (kyotov@cs.cornell.edu)

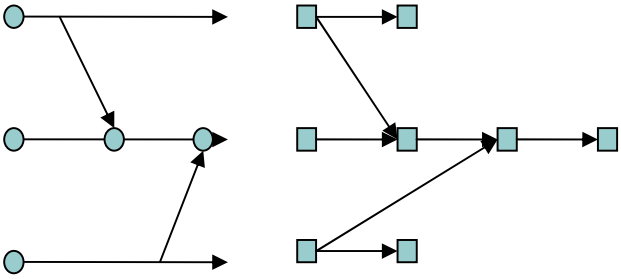
Contents

Introduction

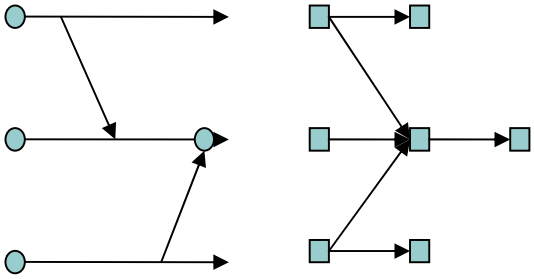
This report briefly presents the results of my project for CS717. This includes two simplified centralized algorithms for fault-tolerance in message passing systems, namely Uncoordinated Check-pointing and (Optimistic) Message Logging. Both algorithms are presented in detail with a somewhat complicated example. Further, they are implemented in C#, for demonstration purposes.

In order to look at Check-pointing and Message Logging together, we need to clear out the issue about any differences between these two approaches. One way to look at Message Logging is that it is really a special case of Check-pointing where we check-point after every message. Essentially every logged message constitutes a "pseudo" check-point.

There is a minor difference between the two concerning their dependence graphs, but we will not worry about it, as it does not affect the presented algorithms. Practically, the difference is that in Message Logging each node of the dependence graph has exactly two predecessors – the previous node in the same process and a node in the process that sent the message, which created this node (state interval), while in Check-pointing we can have higher number of predecessors per single node. Here is the simplest example:



Message Logging



Check-pointing

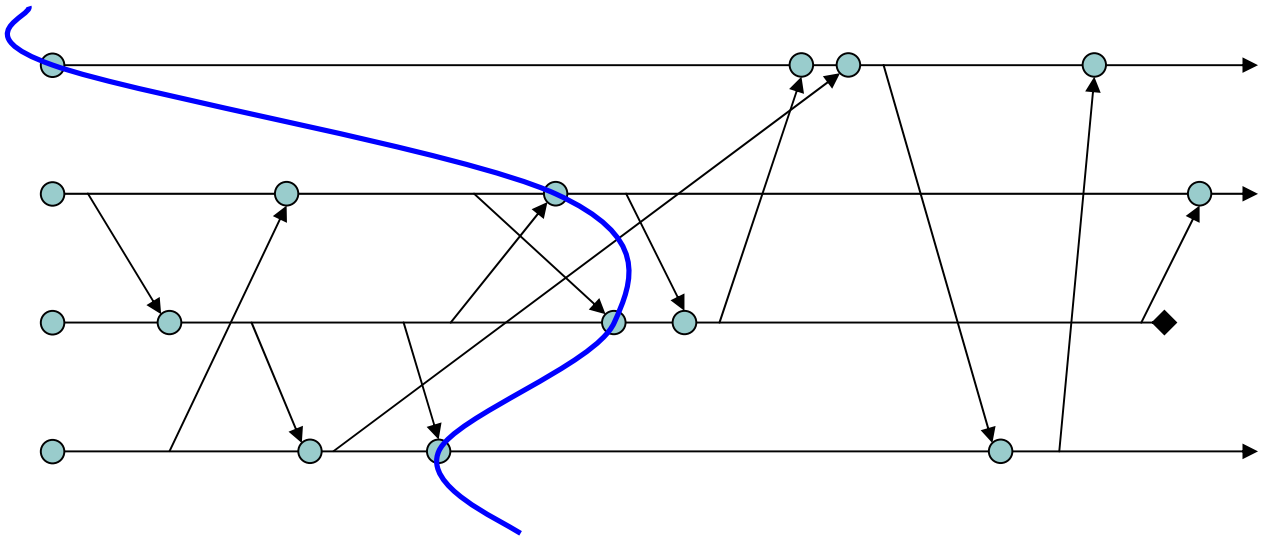
Disclaimer

This report is very concise and not self contained. For example it does not discuss what is a dependence graph, state interval or consistent global state (cut). Further, it emphasizes on finding the best recovery line, while leaving other details on the side. If sufficient interest is expressed in this work, I will expand the report to be more accessible to people outside the course.

Design

Setting

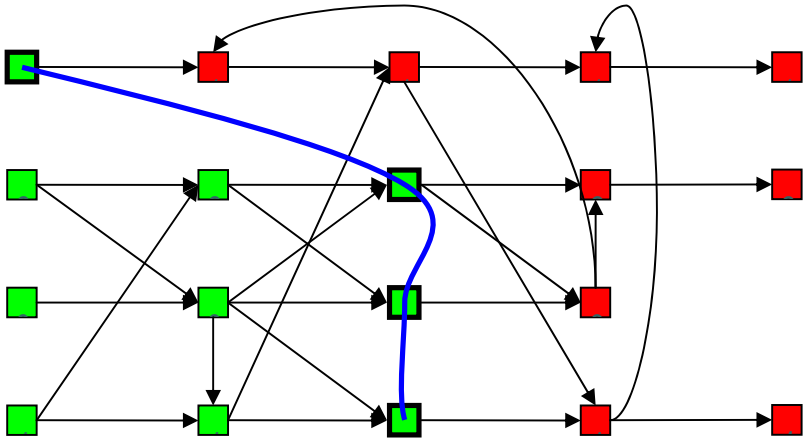
Contained in the project archive is PowerPoint slide show that animates both algorithms step by step on the example depicted below.



The thick blue line here represents the current best recovery line. The diamond indicates that the third processor failed at that point.

Uncoordinated Check-pointing

For the purpose of the Uncoordinated Check-pointing algorithm, we need to construct the Backward Dependence Graph. Intuitively the nodes of this graph are formed by received messages and the message send is associated with the state interval that is instantiated by the last received message.



Here we have colored green all state intervals (nodes) that depend only on other green nodes. Intuitively the algorithm goes as follows: At the beginning all nodes are green. Because message sends are associated with the interval of a previous message receive, we are restricted that our current recovery line cannot contain edges of the dependency graph. Further, no node behind the recovery line should be reachable from a node on the recovery line. Doing a simple graph traversal, we move the recovery line back as much as we need in order to observe these two conditions.

Here is the more formal high-level version of the algorithm:

- Activates on failure
- Modified form of graph reachability
  - Choose your favorite search algorithm
- Overview
  - Color all nodes of G green
  - Let S be the set of all last nodes (process-wise)
  - Repeat
    - Find green node T reachable from a node in S
      - If there is no such node, we are done
      - If the node to the left of T is green, add it to S
      - If T is in S, remove it from S
      - Mark T red

Here is one efficient lower-level implementation:

- 1. Let C be the n-tuple of the last nodes (process-wise)
- 2. Enqueue all nodes in C to Q
- 3. Let V = C
- 4. While Q is non-empty
  - a) Dequeue node T from Q
  - b) For each S in T.next
    - if S.back is green
      - if not S.back in V
        - enqueue S.back to Q

- V = V + S.back
- Update the corresponding element of C to S.back

- else
  - If not S in V
    - enqueue S to Q
    - V = V + S
  - mark S red

Please consult the PowerPoint slide show (slides 11-22) for a running step-by-step example.

This algorithm is easily shown to be equivalent to the one presented in class, based on the discussed papers. Further its clarity and conciseness make it especially useful for implementation.

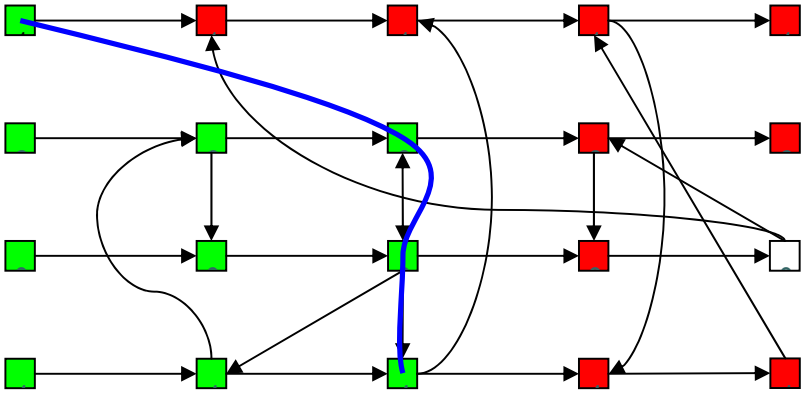
(Optimistic) Message Logging

For Message Logging we would be looking at an incremental algorithm equivalent to the one presented by me in class from the corresponding paper. One way to proceed is to use again the Backward Dependency Graph with the following algorithm:

- Let C be the n-tuple of the first nodes (process-wise)
- Let G contain only the nodes in C
- When a new edge <S,T> arrives
  - If S or T do not exist
    - Create S and/or T and all previous nodes
    - Mark all new nodes red
  - G = G + <S,T>
  - Q = <T>
  - While Q not empty
    - Dequeue T from Q
    - If both predecessors of T are green / yellow
      - Mark T yellow
      - Enqueue all nodes in T.next to Q
  - Check if update of C is possible (yellow to green…)

A running step-by-step example is presented on slides 24-35. This algorithm is somewhat complex and with unclear running time properties, due to the double color updates (first to yellow, than to green). Moreover it is a bit far away from what is presented in the paper.

In order to improve our understanding about the situation, we might want to look at another version of the dependency graph, namely Forward Dependency Graph. The Forward Dependency Graph for our example is provided below. Note that in this version we include a node for the hypothetical state of the failed process.



Now updating the current best recovery line is if not simpler, at least closer to what we have already seen in the paper. When a new edge arrives, we check if it is the case that we can add the edges for all nodes that the destination node depends on. Note that this can involve cycles, which corresponds to updating one row at a time in the matrix version in the paper. We don't need extra colors and obviously there is one-to-one correspondence with the algorithm that we have seen. The running time complexity might not be obvious again, but it is at least evident that the new algorithm is easier to implement and understand. Again, a running step-by-step example is provided on slides 37-48.

Implementation

All the implementation work has been completed in C# more for demonstration purposes than for real use. Everything is encapsulated in a flexible framework, open to further development. Here are some important details:

- **Process** is a conceptual process that knows its number;
- **Message** is a conceptual message that knows its originating and destination processes;
- **Event** is an abstract entity that describes that certain thing happened on a specific process. Event types include Start, Fail, Calculate, Send, and Receive;
- **Node** is a node in a dependence graph.

The demonstrated code (Appendix A) contains fragments to construct Forward and Backward Dependency Graphs (Appendix B). Node names in the output are referring to the names chosen for dependency graph nodes on the slides.

Appendices

Appendix A: C# Source Code

```
using System;
using System.Collections;

namespace Coloring
{
 class Process
 {
```

```
 int n;

 public int number
 {
  get
  {
   return n + 1;
  }
 }

 public string name
 {
  get
  {
   return "p" + n;
  }
 }

 public Process (int n)
 {
  this.n = n;
 }
}

class Message
{
 Process f;
 Process t;

 public Process from
 {
  get
  {
   return f;
  }
 }

 public Process to
 {
  get
  {
   return t;
  }
 }

 public Message (Process f, Process t)
 {
  this.f = f;
  this.t = t;
 }
}

enum EventType
{
 Start,
 Send,
 Receive,
 Calculate,
 Fail
```

```
}

class Event
{
 EventType t;
 Message m;

 public EventType type
 {
  get
  {
   return t;
  }
 }

 public Message message
 {
  get
  {
   return m;
  }
 }

 public Event (EventType t, Message m)
 {
  this.t = t;
  this.m = m;
 }
}

class Node
{
 int p;
 int n;
 Event e;
 ArrayList _s;
 ArrayList _p;

 public ArrayList succ
 {
  get
  {
   return _s;
  }
 }

 public ArrayList pred
 {
  get
  {
   return _p;
  }
 }

 public Node next
 {
  get
  {
   foreach (Node n in succ)
```

```
    if (n.process == process)
     return n;

   return null;
  }
 }

 public Node prev
 {
  get
  {
   foreach (Node n in pred)
    if (n.process == process)
     return n;

   return null;
  }
 }

 public int process
 {
  get
  {
   return p + 1;
  }
 }

 public int number
 {
  get
  {
   return n + 1;
  }
 }

 public string name
 {
  get
  {
   return "n" + process + number;
  }
 }

 public EventType type
 {
  get
  {
   return e.type;
  }
 }

 public Message message
 {
  get
  {
   return e.message;
  }
 }
```

```
  public Node (int p, int n, Event e)
  {
   this.p = p;
   this.n = n;
   this.e = e;

   _s = new ArrayList();
   _p = new ArrayList();
  }
 }

 class Run
 {
  static void Link (Node f, Node t)
  {
   f.succ.Add(t);
   t.pred.Add(f);
  }

  static void UnLink (Node f, Node t)
  {
   f.succ.Remove(t);
   t.pred.Remove(f);
  }

  static Node[] BackwardDependenceGraph (Event[][] h)
  {
   Hashtable t = new Hashtable();

   Node[] n = new Node[h.Length];
   Node[] r = new Node[h.Length];
   int[] hi = new int[h.Length];
   int hc = 0;

   for (int i = 0; i < h.Length; i++)
   {
    n[i] = new Node(i, 0, new Event(EventType.Start, null));
    r[i] = n[i];

    hi[i] = 0;

    hc += h[i].Length;
   }

   for (int i = 0; hc > 0; i = (i + 1) % h.Length)
   {
    Event e = null;

    for (; hi[i] < h[i].Length;)
    {
     e = (Event)h[i][hi[i]];

     if (e.type != EventType.Send)
      break;

     t.Add(e.message, n[i]);

     hi[i]++;
     hc--;
```

```
    }

    if (hi[i] < h[i].Length && (
     e.type == EventType.Fail ||
     e.type == EventType.Calculate ||
     e.type == EventType.Receive && t.Contains(e.message)))
    {
     Node nt = new Node(i, n[i].number, e);

     Link(n[i], nt);

     n[i] = nt;

     if (e.type == EventType.Receive)
      Link((Node)t[e.message], n[i]);

     hi[i]++;
     hc--;
    }
   }

   return r;
  }

  static void FixForward (Node n)
  {
   Node no = null;

   foreach (Node nn in n.succ)
    if (nn.process == n.process)
    {
     FixForward(nn);

     no = nn;

     break;
    }

   if (no == null)
    return;

   foreach (Node nn in (ArrayList)n.succ.Clone())
    if (nn != no)
    {
     UnLink(n, nn);
     Link(no, nn);
    }
  }

  static Node[] ForwardDependenceGraph (Event[][] h)
  {
   Node[] n = BackwardDependenceGraph(h);

   for (int i = 0; i < n.Length; i++)
                FixForward(n[i]);

   return n;
  }
```

```
static void UncoordinatedCheckpointing (Node[] BDG)
{
 Node[] CRL = new Node[BDG.Length];

 Queue Q = new Queue();
 Hashtable V = new Hashtable();

 for (int i = 0; i < BDG.Length; i++)
 {
  Node n = BDG[i];

  while (n.next != null && n.next.type != EventType.Fail)
   n = n.next;

  CRL[i] = n;

  Q.Enqueue(n);
  V.Add(n, null);
 }

 while (Q.Count > 0)
 {
  Node n = (Node)Q.Dequeue();

  Console.WriteLine(n.name + " dequeued!");

  foreach (Node nn in n.succ)
  {
   int p = nn.process;

   Node nne = nn;

   if (CRL[p - 1].number >= nn.number)
   {
    nne = nn.prev;

    Console.WriteLine("C updated " + CRL[p - 1].name + " -> " + nne.name);

    CRL[p - 1] = nne;
   }

   if (!V.Contains(nne))
   {
    Q.Enqueue(nne);
    V.Add(nne, null);

    Console.WriteLine(nne.name + " enqueued!");
   }
  }
 }

 Console.Write("Current Recovery Line: ");

 foreach (Node n in CRL)
  Console.Write(n.name + " ");

 Console.WriteLine();
}
```

```
static void PrintNode (Node n, Hashtable h)
{
 h.Add(n, null);

 foreach (Node nn in n.succ)
 {
  Console.WriteLine(n.name + " -> " + nn.name);

  if (!h.Contains(nn))
   PrintNode(nn, h);
 }
}

static void PrintGraph (Node[] n)
{
 Hashtable h = new Hashtable();

 for (int i = 0; i < n.Length; i++)
  PrintNode(n[i], h);
}

static void Main(string[] args)
{
 Process[] p =
 {
  new Process(0),
  new Process(1),
  new Process(2),
  new Process(3)
 };

 Message[] m =
 {
  new Message(p[1], p[2]), // 00
  new Message(p[3], p[1]), // 01
  new Message(p[2], p[3]), // 02
  new Message(p[3], p[0]), // 03
  new Message(p[2], p[3]), // 04
  new Message(p[2], p[1]), // 05
  new Message(p[1], p[2]), // 06
  new Message(p[1], p[2]), // 07
  new Message(p[2], p[0]), // 08
  new Message(p[0], p[3]), // 09
  new Message(p[3], p[0]), // 10
  new Message(p[2], p[1])  // 11
 };

 Event[][] h =
 {
  new Event[] {
   new Event(EventType.Receive,m[08]),
   new Event(EventType.Receive,m[03]),
   new Event(EventType.Send,  m[09]),
   new Event(EventType.Receive,m[10]),
   new Event(EventType.Calculate,null)
  },
  new Event[] {
   new Event(EventType.Send,  m[00]),
   new Event(EventType.Receive,m[01]),
```

```
      new Event(EventType.Send,  m[06]),
      new Event(EventType.Receive,m[05]),
      new Event(EventType.Send,  m[07]),
      new Event(EventType.Receive,m[11]),
      new Event(EventType.Calculate,null)
     },
     new Event[] {
      new Event(EventType.Receive,m[00]),
      new Event(EventType.Send,  m[02]),
      new Event(EventType.Send,  m[04]),
      new Event(EventType.Send,  m[05]),
      new Event(EventType.Receive,m[06]),
      new Event(EventType.Receive,m[07]),
      new Event(EventType.Send,  m[08]),
      new Event(EventType.Send,  m[11]),
      new Event(EventType.Fail,  null)
     },
     new Event[] {
      new Event(EventType.Send,  m[01]),
      new Event(EventType.Receive,m[02]),
      new Event(EventType.Send,  m[03]),
      new Event(EventType.Receive,m[04]),
      new Event(EventType.Receive,m[09]),
      new Event(EventType.Send,  m[10]),
      new Event(EventType.Calculate,null)
     }
    };

    Node[] BDG = BackwardDependenceGraph(h);

    Console.WriteLine("Backward Dependence Graph");
    PrintGraph(BDG);
    Console.WriteLine();

    Node[] FDG = ForwardDependenceGraph(h);

    Console.WriteLine("Forward Dependence Graph");
    PrintGraph(FDG);
    Console.WriteLine();

    Console.WriteLine("Uncoordinated Checkpointing");
    UncoordinatedCheckpointing(BDG);
    Console.WriteLine();
   }
  }

}
```

Appendix B: Sample Output

Backward Dependence Graph

```
n11 -> n12
n12 -> n13
n13 -> n44
n44 -> n45
n44 -> n14
n14 -> n15
n13 -> n14
```

```
n21 -> n32
n32 -> n42
n42 -> n43
n43 -> n44
n42 -> n13
n32 -> n23
n23 -> n34
n34 -> n35
n34 -> n12
n34 -> n24
n24 -> n25
n23 -> n24
n32 -> n33
n33 -> n34
n32 -> n43
n21 -> n22
n22 -> n23
n22 -> n33
n31 -> n32
n41 -> n22
n41 -> n42

Forward Dependence Graph
n11 -> n12
n12 -> n13
n13 -> n14
n14 -> n15
n14 -> n44
n44 -> n45
n45 -> n14
n21 -> n22
n22 -> n23
n23 -> n24
n24 -> n25
n24 -> n34
n34 -> n35
n35 -> n12
n35 -> n24
n23 -> n33
n33 -> n34
n33 -> n42
n42 -> n43
n43 -> n44
n43 -> n13
n42 -> n22
n33 -> n23
n33 -> n43
n22 -> n32
n32 -> n33
n31 -> n32
```

```
n41 -> n42

Uncoordinated Checkpointing
n15 dequeued!
n25 dequeued!
n34 dequeued!
n35 enqueued!
C updated n15 -> n11
n11 enqueued!
C updated n25 -> n23
n23 enqueued!
n45 dequeued!
n35 dequeued!
n11 dequeued!
n12 enqueued!
n23 dequeued!
C updated n34 -> n33
n33 enqueued!
n24 enqueued!
n12 dequeued!
n13 enqueued!
n33 dequeued!
n24 dequeued!
n13 dequeued!
C updated n45 -> n43
n43 enqueued!
n14 enqueued!
n43 dequeued!
n44 enqueued!
n14 dequeued!
n44 dequeued!
Current Recovery Line: n11 n23 n33 n43
```