Design and Analysis of a Gossip Algorithm

Jamie Brandon

1 Introduction

Gossip (or epidemic) algorithms are an important tool in many peer-to-peer systems. While there are a number of related algorithms that are referred to as gossip algorithms, [14] defines a core common to almost all of these: the peer sampling service (PSS). This service provides each node with a random selection of other nodes to gossip with.

It is argued in [4] that one of the main points in favour of using gossip algorithms to build distributed systems is that their simplicity makes them amenable to formal analysis. Despite this, no formal analysis of a peer sampling service has been produced. Many empirical studies have been conducted [14, 4]. In particular [14] demonstrates that many of the assumptions commonly made about the behaviour of peer sampling services do not hold in practice. It has also been noted [14, 10, 3, 18] that the behaviour of gossip algorithms is often highly sensitive to small changes. Finally, many gossip algorithms have turned out to be surprisingly susceptible to malicious attack [23, 15, 16, 17]. For these reasons it would be beneficial to have a stronger theoretical foundation to work from.

In this dissertation we will present a peer sampling service for which we can make strong theoretical guarantees. Section 2 covers background information. Section 3 describes our general approach and introduces the necessary mathematics. Section 4 describes a centralised algorithm for providing independent, uniformly distributed peer samples. Section 5 discusses balancing the load of the centralised algorithm across multiple nodes. Section 6 introduces a fully distributed version of the previous algorithm. Section 7 investigates the algorithm's resilience to random failure and churn. Section 8 presents a reference implementation and test results. Finally, section 9 presents a conclusion and discusses future extensions.

2 Background

Gossip algorithms have been used for a wide variety of tasks, including overlay construction [26], database replication [9], failure detection [25], resource monitoring [24], distributed search/recommendation [21], reputation tracking [22] and multicast messaging [20].

Recently there has been a flurry of work aimed at rigorous analysis of these algorithms. These can be divided roughly into two groups: those studying the design and theoretical limitations of gossip algorithms operating on a fixed overlay, typically in the context of wireless devices [6, 7], and those attempting to model the behaviour of gossip algorithms on dynamic, unstructured overlays [12, 5, 19, 2, 1]. We are interested here in the latter approach.

Despite the amount of attention it has turned out to be very difficult to analyse such algorithms. There have been a number of successes in modelling aspects of existing algorithms [5, 19, 1] but these have required a substantial amount of effort to demonstrate a small number of useful properties. We will instead produce an algorithm which is designed with ease of modelling in mind.

3 Approach

We will argue that there are several properties of existing gossip algorithms that makes them fundamentally difficult to analyse.

The first of these is the use of fixed delays between actions. This causes complex interleaving of actions between nodes. For example, in [1] the author begins modelling a simple gossip algorithm by making the simplifying assumption that view updates are scheduled randomly. The resulting predictions are shown to be incorrect and the author is forced to move to a more complicated mathematical model. We will demonstrate that it is possible to arrange that actions are scheduled uniformly at random between nodes and this makes modelling the system much simpler.

The second is conflation of subsystems within an algorithm. Most gossip algorithms are presented as a single monolithic algorithm whereas it would be simpler to consider them as separate layers performing different functions. For example, in Cyclon and similar algorithms the network view and the peer sampling service are combined in a single data structure and both are updated simultaneously. It has been recognised [14, 12] that the separation of these jobs is a natural simplification. Here we will concentrate entirely on the peer sampling service as a system providing a stream of random peer samples over time. Other subsystems, such as the network view, can be built as consumers of this stream.

The third, and somewhat less well defined, problem is that interactions between nodes are too complicated. We will show that the view shuffling used in most gossip algorithms is unnecessary - a peer sampling service can be produced without being aware of the network view at all.

Our inspiration for this algorithm comes from a natural example: radioactive decay. A group of radioactive atoms decays in a uniformly random order. That is, each atom has an equal probability to be the next atom to decay. This does not require any interaction between the atoms. It is a simple consequence of the exponential distribution of decay times.

The exponential distribution is governed by the probability distribution function $f(x) = \lambda e^{-\lambda x}$. This distribution has a number of interesting properties. It is memoryless: given $E \sim \text{Exp}(\lambda)$ we can prove that $P(E > a + b \mid E > a) = P(E > b)$. Given two such variables $E_1 \sim \text{Exp}(\lambda)$ and $E_2 \sim \text{Exp}(\mu)$ we can prove that $\min(E_1, E_2) \sim \text{Exp}(\lambda + \mu)$ and $P(E_1 < E_2) = \lambda/(\mu + \lambda)$. These properties can be recursively extended to more than two variables.

We can see now why atoms decay randomly. Since they all have the same distribution they all have the same probability to be the first to decay. Once one atom has decayed the memoryless property means that we can 'reset' all the decay times.

Atoms can only decay once. If we allow repeated decay we obtain a Poisson process. A Poisson process is a sequence of exponential decay times. More formally, given decay times $S_i \sim \text{Exp}(\lambda)$, the Poisson process with rate λ is the sequence $T_i = \sum_{k=0}^{i} S_k$. It is often useful to associate a set of random events E_i with the process, so that the event E_i is considered to have occurred at time T_i.

Like the exponential distribution this process is memoryless - it looks the same starting from any point. It also has two properties which we will use repeatedly. The merging property governs interleaving two different Poisson

processes into a single process. The thinning property governs splitting a single Poisson process into two independent processes.

Merging. *Let $U = \{U_i | i \geq 0\}$ and $V = \{V_i | i \geq 0\}$ be independent Poisson processes with rates λ and μ respectively. WLOG let $U_0 < V_0$. Define the merging $T = merge(U, V)$ of these processes recursively as follows: $T_0 = U_0$, $\{T_i \mid i > 0\} = merge(\{U_i \mid i > 0\}, V)$. Then T is a Poisson process with rate $\lambda + \mu$. Define E_i to be 0 if T_i was drawn from U and 1 otherwise. Then the random variables $\{E_i \mid i \geq 0\}$ are independent and $E_i \sim Bin(\lambda/(\lambda + \mu))$.*

Thinning. *Let T be a Poisson process with rate λ. Given a set E_i of independent random binary variables $E_i \sim Bin(p)$, define $U = \{T_i \mid E_i = 0,\ i \geq 0\}$ and $V = \{T_i \mid E_i = 1,\ i \geq 0\}$. Then U and V are independent Poisson process with rates λp and $\lambda(1 - p)$ respectively.*

Both the merging and thinning properties can be applied recursively to deal with more than two Poisson processes.

4 Implementing a uniform PSS

We can apply these ideas to a simple, centralised peer sampling service. Suppose we have a network consisting of n nodes.

Pick some root node R, whose address is known to all other nodes. Every other node behaves as an independent Poisson process sending messages to the root node at a rate λ. Let R_i be the sender of the ith message to arrive at the root node. By the merging property the sequence R_i forms a possion process of rate $n\lambda$ and each R_i is an independent uniform random choice from the set of all nodes.

On receiving each message R_i, the root node will send the address of R_{i-1} to R_i (the message R_0 is ignored). Let N_i be the ith such message to arrive at node N from the root R. Let T_i^N be the index in R of this message i.e.

$$S_{T_i^N} = N$$
$$S_{T_i^N - 1} = N_i$$

For each node N we use the sequence of addresses N_i as the peer samples for N.

Theorem. *For each node N the peer samples N_i are independent*

Proof. For any node M

$$\begin{aligned} &P(N_i = M |\ N_0 = n_0, ..., N_{i-1} = n_{i-1}) \\ &\quad = P(S_{T_i^N - 1} = M |\ S_{T_0^N - 1} = n_0, ..., S_{T_{i-1}^N - 1} = n_{i-1}) \\ &\quad = P(S_{T_i - 1} = M) \text{ because } S_i \text{ are independent} \end{aligned}$$

□

Theorem. *For each node N the peer samples $N_i,\ i > 0$ are uniformly distributed over the set of all nodes (note that N_0 is not uniformly distributed since it can only be equal to N when $S_0 = S_1 = N$)*

Proof. Consider the peer sample N_i. Define $K = T_i^N - T_{i-1}^N$.

$$
\begin{aligned}
&P(N_i = N) \\
&\quad = \sum_{k>0} P(K = k) P(S_{T_i^N - 1} = N | K = k) \\
&\quad = P(K = 1) P(S_{T_i^N - 1} = N | K = 1) \\
&\qquad\qquad \textit{because } \forall k{>}1 \;\; P(S_{T_i^N - 1} = M | K = k) = 0 \\
&\quad = \frac{1}{n}\, 1
\end{aligned}
$$

For $M \neq N$:

$$
\begin{aligned}
&P(N_i = M) \\
&\quad = P(S_{T_i - 1} = M \\
&\quad = \sum_{k>0} P(K = k) P(S_{T_i^N - 1} = M | K = k) \\
&\quad = \sum_{k>1} P(K = k) P(S_{T_i^N - 1} = M | K = k) \\
&\qquad\qquad \textit{because } P(S_{T_i^N - 1} = M | K = 1) = 0 \\
&\quad = \sum_{k>1} P(K = k) P(S_{T_i^N - 1} = M | K = k, S_{T_i^N - 1} \neq N) \\
&\qquad\qquad \textit{because } T_{i-1}^N = T_i^N - K < T_i^N - 1 < T_i^N \\
&\quad = \sum_{k>1} P(K = k) (\frac{1}{n-1}) \\
&\qquad\qquad \textit{because } S_i \textit{ are independent} \\
&\quad = \frac{1}{n-1} \sum_{k>1} P(K = k) \\
&\quad = \frac{1}{n-1} P(K > 1) \\
&\quad = \frac{1}{n-1} (1 - \frac{1}{n}) \\
&\quad = \frac{1}{n}
\end{aligned}
$$

□

The system as a whole forms a continuous-time Markov chain (CTMC). Using the PRISM model checker we can provide an explicit and unambigous description of this algorithm and verify the desired properties.

```
ctmc

const int n = 3;
const double lambda = 1.0;


module poppi
  root : [0..n-1] init 0;

  node0 : [0..n-1] init 0;
  node1 : [0..n-1] init 0;
  node2 : [0..n-1] init 0;

  //node0 contacts the root and receives a new peer
      selection
  [] true -> lambda:(root'=0)&(node0'=root);

  //node1 contacts the root and receives a new peer
      selection
  [] true -> lambda:(root'=1)&(node1'=root);

  //node2 contacts the root and receives a new peer
      selection
  [] true -> lambda:(root'=2)&(node2'=root);

endmodule
```

For the sake of clarity, in this section we will only present example models for networks with 3 nodes and describe the properties which need to be tested. More complete details on model generation and the methods for testing various properties are described in Appendix A.

In the above model the variable 'root' represents the last node to send a message to the root node and the variables 'node0', 'node1' and 'node2' represent the last peer sample at their respective nodes.

Suppose we wanted to test the claim that each N_i is uniformly distributed for $i > 0$. This turns out to be very difficult in this model. We could test $P(N_i = M)$ for any fixed point in time and check that the value is the same for each node M:

```
P=? [ F[T,T] node0=0 ]
P=? [ F[T,T] node0=1 ]
P=? [ F[T,T] node0=2 ]
```

Or we could test the same property in the steady-state:

```
S=? [ node0=0 ]
S=? [ node0=1 ]
S=? [ node0=2 ]
```

While these are useful properties to have neither of them are quite equivalent to saying that N_i are uniformly distributed. The problem is that each N_i is associated with a transition whereas the properties above are associated with points in time. It is awkward to specify PRISM properties relating to transitions when using a CTMC model. A possible workaround is to use transition rewards but this brings its own problems. Transition rewards have no meaning in the

steady-state and cannot be calculated by the PRISM simulator, which makes working with larger models impossible.

A simpler solution is to ignore transition times altogether and work with the jump process - the discrete-time Markov chain (DTMC) corresponding to the transitions of the CTMC. The translation is straightforward:

```
dtmc

const int n = 3;

module next
  next : [0..n-1] init 0;

  [next] true -> (1/n):(next'=0) + (1/n):(next'=1) +
      (1/n):(next'=2);
endmodule


module poppi
  root : [0..n-1] init 0;

  node0 : [0..n-1] init 0;
  node1 : [0..n-1] init 0;
  node2 : [0..n-1] init 0;

  //node0 contacts the root and receives a new peer
      selection
  [next] (next=0) -> (root'=0)&(node0'=root);

  //node1 contacts the root and receives a new peer
      selection
  [next] (next=1) -> (root'=1)&(node1'=root);

  //node2 contacts the root and receives a new peer
      selection
  [next] (next=2) -> (root'=2)&(node2'=root);

endmodule


module poppi_past
  next_past : [0..n-1] init 0;

  node0_past : [0..n-1] init 0;
  node1_past : [0..n-1] init 0;
  node2_past : [0..n-1] init 0;

  [next] true -> (next_past'=next) & (node0_past'=node0)
     & (node1_past'=node1) & (node2_past'=node2);
endmodule
```

Notice that the choice of which node will next send a message to the root has been made explicit. Also the 'Past' module has been added, which tracks the last value of each variable. This makes it easy to identify peer samples. For example, if next_past is 0 then node0 is a new peer sample, replacing the previous sample node0_past.

Now we can directly test that the distribution of peer samples at a given node is uniform in the steady-state. The following must all be equal:

```
S=? [ next_past=0 & node0=0 ]
S=? [ next_past=0 & node0=1 ]
S=? [ next_past=0 & node0=2 ]
```

Similarly, we can test that each peer sample is independent of the previous peer sample by checking that the following are equal:

```
S=? [ next_past=0 & node0=0 & node_past=0 ]
S=? [ next_past=0 & node0=0 & node_past=1 ]
S=? [ next_past=0 & node0=0 & node_past=2 ]

S=? [ next_past=0 & node0=0 & node_past=0 ]
S=? [ next_past=0 & node0=1 & node_past=1 ]
S=? [ next_past=0 & node0=2 & node_past=2 ]

S=? [ next_past=0 & node0=0 & node_past=0 ]
S=? [ next_past=0 & node0=1 & node_past=1 ]
S=? [ next_past=0 & node0=2 & node_past=2 ]
```

For good measure, we can check in both models that the latest peer samples at any two given nodes are independent by testing e.g.:

```
S=? [ node0=0 & node1=0 ]
S=? [ node0=0 & node1=1 ]
S=? [ node0=0 & node1=2 ]

S=? [ node0=1 & node1=0 ]
S=? [ node0=1 & node1=1 ]
S=? [ node0=1 & node1=2 ]

S=? [ node0=2 & node1=0 ]
S=? [ node0=2 & node1=1 ]
S=? [ node0=2 & node1=2 ]
```

This algorithm is easy to reason about and provides strong guarantees about the distribution of peer samples. It is also simple enough that we can model it directly in PRISM and verify a subset of the properties that were proved analytically. Unfortunately it introduces a single point of failure at the root node, which must sustain a load proportional to the number of nodes in the network. This is obviously unacceptable for a large network. The next sections will describe how to remove this point of failure.

5 Spreading the load

A simple aproach to improving the previous algorithm is to use more than one root node. We can replace the single root node with a set of root nodes $R^0..R^k$ which are known to every node. Each of the new root nodes behave identically to the root node in the previous algorithm. For each message the sending node will choose a root node uniformly at random to contact.

Intuitively, this has the same steady-state properties as the previous algorithm. The argument is as follows: By the thinning property we can consider the messages arriving at each root node to be independent Poisson processes each with rate $n\lambda/k$. Then each node receives k independent peer sampling processes (one from each root node) each with rate λ/k and with peer samples which are independent and uniformly distributed as proved for the previous algorithm. By

the merging property we can combine these peer sampling processes to form a single process with the same properties.

One change we should be careful to note is that in the previous algorithm the first peer sample N_0 was not uniformly distributed. Since we now have k of these processes being interleaved we cannot guarantee when the first peer sample from each node will occur. We can guarantee that in the worst case each node has to send two messages to each root node before all future peer samples at that node will be uniformly distributed (the first message might be the first message at that node, in which case it will not generate a peer sample). Define I_N to be the point at which at least two messages have been sent from N to each peer node.

Theorem. $E(I_N) \leq \frac{2kn}{n-k+1}$

Proof. Define J_N^k to be the point at which node N has sent at least one message to each of k different root nodes.

$$\begin{aligned}
E(J_N^k) &= E(J_N^k + (J_N^{k-1} - J_N^{k-1}) + ... + (J_N^1 - J_N^1)) \\
&= E(J_N^k - J_N^{k-1}) + E(J_N^{k-1} - J_N^{k-2}) + ... + E(J_N^2 - J_N^1) + E(J_N^1) \\
&= \frac{n}{n-k+1} + \frac{n}{n-k+2} + ... + \frac{n}{n-1} + \frac{n}{n} \\
&\leq k\frac{n}{n-k+1} \\
E(I_N) \ &\leq \ 2E(J_N^k) \ \leq \ 2k\frac{n}{n-k+1}
\end{aligned}$$

□

Note that even before I_N is reached the peer samples at N will still be uniformly distributed across all nodes except N and the expected number of messages sent before the first peer sample is generated is only $\frac{k}{n}$.

We can model this new algorithm in PRISM:

```
ctmc

const int n = 3;
const int k = 2;
const double lambda = 1.0;


module poppi
  root0 : [0..n-1] init 0;
  root1 : [0..n-1] init 0;

  node0 : [0..n-1] init 0;
  node1 : [0..n-1] init 0;
  node2 : [0..n-1] init 0;

  //node0 contacts the root and receives a new peer
      selection
  [] true -> (lambda/k):(root0'=0)&(node0'=root0) +
      (lambda/k):(root1'=0)&(node0'=root1);
```

```
  //node1 contacts the root and receives a new peer
      selection
  [] true -> (lambda/k):(root0'=1)&(node1'=root0) +
      (lambda/k):(root1'=1)&(node1'=root1);

  //node2 contacts the root and receives a new peer
      selection
  [] true -> (lambda/k):(root0'=2)&(node2'=root0) +
      (lambda/k):(root1'=2)&(node2'=root1);

endmodule
```

Again, many properties are easier to prove using the jump process.

```
dtmc

const int n = 3;
const int k = 2;

module next
  next : [0..n-1] init 0;

  [next] true -> (1/n):(next'=0) + (1/n):(next'=1) +
      (1/n):(next'=2);
endmodule


module poppi
  root0 : [0..n-1] init 0;
  root1 : [0..n-1] init 0;

  node0 : [0..n-1] init 0;
  node1 : [0..n-1] init 0;
  node2 : [0..n-1] init 0;

  //node0 contacts the root and receives a new peer
      selection
  [next] (next=0) -> (1/k):(root0'=0)&(node0'=root0) +
      (1/k):(root1'=0)&(node0'=root1);

  //node1 contacts the root and receives a new peer
      selection
  [next] (next=1) -> (1/k):(root0'=1)&(node1'=root0) +
      (1/k):(root1'=1)&(node1'=root1);

  //node2 contacts the root and receives a new peer
      selection
  [next] (next=2) -> (1/k):(root0'=2)&(node2'=root0) +
      (1/k):(root1'=2)&(node2'=root1);

endmodule


module poppi_past
  next_past : [0..n-1] init 0;

  node0_past : [0..n-1] init 0;
  node1_past : [0..n-1] init 0;
  node2_past : [0..n-1] init 0;
```

```
  [next] true -> (next_past'=next) & (node0_past'=node0)
      & (node1_past'=node1) & (node2_past'=node2);
endmodule
```

We can reuse all the properties tested for the original algorithm without change and, as expected, all of them still hold.

6 Inside out

Adding more root nodes spreads out the load and makes the network more resilient but this still leaves us with a predetermined set of root nodes to which we cannot add or remove nodes. Examining the proof above, the only reason that the set of root nodes needs to be known is in order to make a random choice from it. These choices are required to be uniformly distributed and independent. If we had a peer sampling service for the root nodes we would not need to know their addresses in advance and we could add and remove root nodes.

The solution is to feed the output of the algorithm back in to itself. Each node can also act as a root node. The system is bootstrapped from a known set of root nodes so that each node has generated at least one peer sample. Then these independent uniform peer samples are used to choose which node to contact next to receive a new peer sample.

This seems intuitively correct but there are a number of subtle details that have been glossed over. For example, is the output of the peer sampling service independent of the input choices? Will uniform sampling still hold if some nodes start acting as root nodes before others have bootstrapped? We can check our intuition with PRISM.

```
ctmc

const int n = 3;
const int k = 3;
const double lambda = 1.0;


module poppi
  root0 : [0..n-1] init 0;
  root1 : [0..n-1] init 0;
  root2 : [0..n-1] init 0;

  node0 : [0..n-1] init 0;
  node1 : [0..n-1] init 0;
  node2 : [0..n-1] init 0;

  //node0 contacts the root and receives a new peer
      selection
  [] (node0=0) -> lambda:(root0'=0)&(node0'=root0);
  [] (node0=1) -> lambda:(root1'=0)&(node0'=root1);
  [] (node0=2) -> lambda:(root2'=0)&(node0'=root2);

  //node1 contacts the root and receives a new peer
      selection
  [] (node1=0) -> lambda:(root0'=1)&(node1'=root0);
  [] (node1=1) -> lambda:(root1'=1)&(node1'=root1);
```

```
  [] (node1=2) -> lambda:(root2'=1)&(node1'=root2);

  //node2 contacts the root and receives a new peer
      selection
  [] (node2=0) -> lambda:(root0'=2)&(node2'=root0);
  [] (node2=1) -> lambda:(root1'=2)&(node2'=root1);
  [] (node2=2) -> lambda:(root2'=2)&(node2'=root2);

endmodule
```

Again, it is useful to also work with the jump process.

```
dtmc

const int n = 3;
const int k = 3;

module next
  next : [0..n-1] init 0;

  [next] true -> (1/n):(next'=0) + (1/n):(next'=1) +
      (1/n):(next'=2);
endmodule


module poppi
  root0 : [0..n-1] init 0;
  root1 : [0..n-1] init 0;
  root2 : [0..n-1] init 0;

  node0 : [0..n-1] init 0;
  node1 : [0..n-1] init 0;
  node2 : [0..n-1] init 0;

  //node0 contacts the root and receives a new peer
      selection
  [next] (next=0)&(node0=0) -> (root0'=0)&(node0'=root0);
  [next] (next=0)&(node0=1) -> (root1'=0)&(node0'=root1);
  [next] (next=0)&(node0=2) -> (root2'=0)&(node0'=root2);

  //node1 contacts the root and receives a new peer
      selection
  [next] (next=1)&(node1=0) -> (root0'=1)&(node1'=root0);
  [next] (next=1)&(node1=1) -> (root1'=1)&(node1'=root1);
  [next] (next=1)&(node1=2) -> (root2'=1)&(node1'=root2);

  //node2 contacts the root and receives a new peer
      selection
  [next] (next=2)&(node2=0) -> (root0'=2)&(node2'=root0);
  [next] (next=2)&(node2=1) -> (root1'=2)&(node2'=root1);
  [next] (next=2)&(node2=2) -> (root2'=2)&(node2'=root2);

endmodule


module poppi_past
  next_past : [0..n-1] init 0;

  node0_past : [0..n-1] init 0;
  node1_past : [0..n-1] init 0;
```

```
  node2_past : [0..n-1] init 0;

  [next] true -> (next_past'=next) & (node0_past'=node0)
     & (node1_past'=node1) & (node2_past'=node2);
endmodule
```

PRISM quickly reveals that there is a problem with this algorithm. We find, for both the CTMC and the DTMC:

```
S=? [ node0=0 ] = 0.31186...
S=? [ node0=1 ] = 0.34407...
S=? [ node0=2 ] = 0.34407...
```

This is well outside the bounds of numerical error. Clearly the algorithm is flawed.

The algorithm assumes that the output of the peer sampling service is independent of the input choices. Unfortunately, there is a small dependency between the two caused by the fact that not all states in the model are reachable from the initial state. For example, the state where node0=0, node1=1, node2=2, root0=0, root1=1, root2=2 is not reachable from any other state nor can any state be reached from there. Knowing the state of node0 therefore gives some knowledge about the state of node1 and node2. When building the model PRISM reports the extent of the problem:

```
SCCs: 5, BSCCs: 5, non-BSCC states: 0
BSCC sizes: 1:683 2:15 3:15 4:15 5:1
```

This is reminiscent of the random surfer problem [8], in which a person surfs the web by randomly following hyperlinks. The resulting Markov chain has the same structure as the underlying hyperlink graph. If the hyperlink graph is not connected or if it has periodic states then the resulting Markov chain will not have a well defined steady-state. The solution is to add an arbitrarily small probability that the user will jump to a page selected uniformly at random. This transition unifies unconnected states and removes periodicity. Our problem can be fixed in a similar manner by adding an arbitrarily small probability of contacting one of the known root nodes.

```
ctmc

const int n = 3;
const int k = 3;
const double lambda = 1.0;

const double mu = 0.01;


module poppi
  root0 : [0..n-1] init 0;
  root1 : [0..n-1] init 0;
  root2 : [0..n-1] init 0;

  node0 : [0..n-1] init 0;
  node1 : [0..n-1] init 0;
  node2 : [0..n-1] init 0;
```

```
  //node0 contacts the root and receives a new peer
      selection
  [] (node0=0) -> lambda:(root0'=0)&(node0'=root0);
  [] (node0=1) -> lambda:(root1'=0)&(node0'=root1);
  [] (node0=2) -> lambda:(root2'=0)&(node0'=root2);
  [] true -> mu:(root0'=0)&(node0'=root0);

  //node1 contacts the root and receives a new peer
      selection
  [] (node1=0) -> lambda:(root0'=1)&(node1'=root0);
  [] (node1=1) -> lambda:(root1'=1)&(node1'=root1);
  [] (node1=2) -> lambda:(root2'=1)&(node1'=root2);
  [] true -> mu:(root0'=1)&(node1'=root0);

  //node2 contacts the root and receives a new peer
      selection
  [] (node2=0) -> lambda:(root0'=2)&(node2'=root0);
  [] (node2=1) -> lambda:(root1'=2)&(node2'=root1);
  [] (node2=2) -> lambda:(root2'=2)&(node2'=root2);
  [] true -> mu:(root0'=2)&(node2'=root0);

endmodule
```

Now all the states are reachable from the initial state.

```
SCCs: 1, BSCCs: 1, non-BSCC states: 0
BSCC sizes: 1:729
```

Translating to the jump process is still straightforward.

```
dtmc

const int n = 3;
const int k = 3;

const double lambda = 1.0;
const double mu = 0.01;
const double node = lambda / (mu + lambda);
const double default = mu / (mu + lambda);

module next
  next : [0..n-1] init 0;

  [next] true -> (1/n):(next'=0) + (1/n):(next'=1) +
      (1/n):(next'=2);
endmodule


module poppi
  root0 : [0..n-1] init 0;
  root1 : [0..n-1] init 0;
  root2 : [0..n-1] init 0;

  node0 : [0..n-1] init 0;
  node1 : [0..n-1] init 0;
  node2 : [0..n-1] init 0;

  //node0 contacts the root and receives a new peer
      selection
```

```
  [next] (next=0)&(node0=0) ->
      node:(root0'=0)&(node0'=root0) +
      default:(root0'=0)&(node0'=root0);
  [next] (next=0)&(node0=1) ->
      node:(root1'=0)&(node0'=root1) +
      default:(root0'=0)&(node0'=root0);
  [next] (next=0)&(node0=2) ->
      node:(root2'=0)&(node0'=root2) +
      default:(root0'=0)&(node0'=root0);

  //node1 contacts the root and receives a new peer
      selection
  [next] (next=1)&(node1=0) ->
      node:(root0'=1)&(node1'=root0) +
      default:(root0'=1)&(node1'=root0);
  [next] (next=1)&(node1=1) ->
      node:(root1'=1)&(node1'=root1) +
      default:(root0'=1)&(node1'=root0);
  [next] (next=1)&(node1=2) ->
      node:(root2'=1)&(node1'=root2) +
      default:(root0'=1)&(node1'=root0);

  //node2 contacts the root and receives a new peer
      selection
  [next] (next=2)&(node2=0) ->
      node:(root0'=2)&(node2'=root0) +
      default:(root0'=2)&(node2'=root0);
  [next] (next=2)&(node2=1) ->
      node:(root1'=2)&(node2'=root1) +
      default:(root0'=2)&(node2'=root0);
  [next] (next=2)&(node2=2) ->
      node:(root2'=2)&(node2'=root2) +
      default:(root0'=2)&(node2'=root0);

endmodule


module poppi_past
  next_past : [0..n-1] init 0;

  node0_past : [0..n-1] init 0;
  node1_past : [0..n-1] init 0;
  node2_past : [0..n-1] init 0;

  [next] true -> (next_past'=next) & (node0_past'=node0)
     & (node1_past'=node1) & (node2_past'=node2);
endmodule
```

Reassuringly, all of our tests now pass and we have a simple, distributed peer sampling service with strong guarantees on the distribution of peer samples.

7 Dealing with failure

We will consider two forms of failure: dropping messages and crashing nodes. In both cases we will respond to failure by falling back to the centralised algorithm i.e. setting the new peer sample to the address of one of the known root nodes. This is a naive strategy but it has the advantage of being simple.

We will start by modelling message dropping. Assume that every message has a probability p of being dropped, independently of every other message.

Each transition in our algorithm requires two messages - one message to request a sample from another node and one message for the reply. If either of these is dropped the sending node will timeout while waiting for the reply. We will assume that the timeout is small enough relative to $1/\lambda$ that we can consider the failure to be an instantaneous transition.

```
ctmc

const int n = 3;
const int k = 3;
const double lambda = 1.0;

const double mu = 0.01;
const double p = 0.1;


module poppi
  root0 : [0..n-1] init 0;
  root1 : [0..n-1] init 0;
  root2 : [0..n-1] init 0;

  node0 : [0..n-1] init 0;
  node1 : [0..n-1] init 0;
  node2 : [0..n-1] init 0;

  //node0 contacts the root and receives a new peer
      selection
  [] (node0=0) ->
      (1-p)*(1-p)*lambda:(root0'=0)&(node0'=root0) +
      (1-p)*p*lambda:(root0'=0)&(node0'=0) +
      p*lambda:(node0'=0);
  [] (node0=1) ->
      (1-p)*(1-p)*lambda:(root1'=0)&(node0'=root1) +
      (1-p)*p*lambda:(root1'=0)&(node0'=0) +
      p*lambda:(node0'=0);
  [] (node0=2) ->
      (1-p)*(1-p)*lambda:(root2'=0)&(node0'=root2) +
      (1-p)*p*lambda:(root2'=0)&(node0'=0) +
      p*lambda:(node0'=0);
  [] true -> (1-p)*(1-p)*mu:(root0'=0)&(node0'=root0) +
      (1-p)*p*mu:(root0'=0);

  //node1 contacts the root and receives a new peer
      selection
  [] (node1=0) ->
      (1-p)*(1-p)*lambda:(root0'=1)&(node1'=root0) +
      (1-p)*p*lambda:(root0'=1)&(node1'=0) +
      p*lambda:(node1'=0);
  [] (node1=1) ->
      (1-p)*(1-p)*lambda:(root1'=1)&(node1'=root1) +
      (1-p)*p*lambda:(root1'=1)&(node1'=0) +
      p*lambda:(node1'=0);
  [] (node1=2) ->
      (1-p)*(1-p)*lambda:(root2'=1)&(node1'=root2) +
      (1-p)*p*lambda:(root2'=1)&(node1'=0) +
      p*lambda:(node1'=0);
  [] true -> (1-p)*(1-p)*mu:(root0'=1)&(node1'=root0) +
      (1-p)*p*mu:(root0'=1);

  //node2 contacts the root and receives a new peer
```

```
    selection
[] (node2=0) ->
    (1-p)*(1-p)*lambda:(root0'=2)&(node2'=root0) +
    (1-p)*p*lambda:(root0'=2)&(node2'=0) +
    p*lambda:(node2'=0);
[] (node2=1) ->
    (1-p)*(1-p)*lambda:(root1'=2)&(node2'=root1) +
    (1-p)*p*lambda:(root1'=2)&(node2'=0) +
    p*lambda:(node2'=0);
[] (node2=2) ->
    (1-p)*(1-p)*lambda:(root2'=2)&(node2'=root2) +
    (1-p)*p*lambda:(root2'=2)&(node2'=0) +
    p*lambda:(node2'=0);
[] true -> (1-p)*(1-p)*mu:(root0'=2)&(node2'=root0) +
    (1-p)*p*mu:(root0'=2);

endmodule
```

For a model with 5 nodes and with $p = 0.1$ we find:

```
S=? [node1=0] = 0.348...
S=? [node1=1] = 0.163...
S=? [node1=2] = 0.163...
S=? [node1=3] = 0.163...
S=? [node1=4] = 0.163...
```

We can see that a high failure rate increases the load on the known root nodes (in this case, node0). This damages the distribution of peer samples. However if we assume that most algorithms consuming the peer samples will not require the known root nodes then we can filter them out. The remaining peer samples are still guaranteed to be uniformly distributed. All the remaining tests pass when references to node0, the known root node, are removed.

Next we need to consider crashing nodes. Given that this algorithm does not require any special behaviour on leaving the network, modelling node failure is very similar to modelling churn in the network. Allow each node in the network to turn off and on at some low rate ϵ. We will represent this by setting the node variable to -1. Sending a message to a node which is in this state will of course fail. When the node turns on it will have lost its previous state and will take the default initial value again.

```
ctmc

const int n = 3;
const int k = 3;
const double lambda = 1.0;

const double mu = 0.01;
const double epsilon = 0.01;


module poppi
  root0 : [-1..n-1] init 0;
  root1 : [-1..n-1] init 0;
  root2 : [-1..n-1] init 0;

  node0 : [-1..n-1] init 0;
  node1 : [-1..n-1] init 0;
```

```
node2 : [-1..n-1] init 0;

//node0 contacts the root and receives a new peer
    selection
[] (node0=0)&(root0!=-1) ->
    lambda:(root0'=0)&(node0'=root0);
[] (node0=1)&(root1!=-1) ->
    lambda:(root1'=0)&(node0'=root1);
[] (node0=2)&(root2!=-1) ->
    lambda:(root2'=0)&(node0'=root2);
[] (node0=0)&(root0=-1) -> lambda:(node0'=0);
[] (node0=1)&(root1=-1) -> lambda:(node0'=0);
[] (node0=2)&(root2=-1) -> lambda:(node0'=0);
[] (root0!=-1) -> mu:(root0'=0)&(node0'=root0);
[] (root0=-1) -> epsilon:(root0'=0)&(node0'=0);
[] (root0!=-1) -> epsilon:(root0'=-1)&(node0'=-1);

//node1 contacts the root and receives a new peer
    selection
[] (node1=0)&(root0!=-1) ->
    lambda:(root0'=1)&(node1'=root0);
[] (node1=1)&(root1!=-1) ->
    lambda:(root1'=1)&(node1'=root1);
[] (node1=2)&(root2!=-1) ->
    lambda:(root2'=1)&(node1'=root2);
[] (node1=0)&(root0=-1) -> lambda:(node1'=0);
[] (node1=1)&(root1=-1) -> lambda:(node1'=0);
[] (node1=2)&(root2=-1) -> lambda:(node1'=0);
[] (root1!=-1) -> mu:(root0'=1)&(node1'=root0);
[] (root1=-1) -> epsilon:(root1'=0)&(node1'=0);
[] (root1!=-1) -> epsilon:(root1'=-1)&(node1'=-1);

//node2 contacts the root and receives a new peer
    selection
[] (node2=0)&(root0!=-1) ->
    lambda:(root0'=2)&(node2'=root0);
[] (node2=1)&(root1!=-1) ->
    lambda:(root1'=2)&(node2'=root1);
[] (node2=2)&(root2!=-1) ->
    lambda:(root2'=2)&(node2'=root2);
[] (node2=0)&(root0=-1) -> lambda:(node2'=0);
[] (node2=1)&(root1=-1) -> lambda:(node2'=0);
[] (node2=2)&(root2=-1) -> lambda:(node2'=0);
[] (root2!=-1) -> mu:(root0'=2)&(node2'=root0);
[] (root2=-1) -> epsilon:(root2'=0)&(node2'=0);
[] (root2!=-1) -> epsilon:(root2'=-1)&(node2'=-1);

endmodule
```

For a network with 5 nodes and $\epsilon = 0.01$ we find:

```
S=? [node1=0] = 0.152...
S=? [node1=1] = 0.150...
S=? [node1=2] = 0.065...
S=? [node1=3] = 0.065...
S=? [node1=4] = 0.065...
```

Again, falling back to the centralised algorithm increases load on the known root node. We also find that each node has a bias towards itself because it can never fail to contact itself. Similarly to the last model, we can still guarantee uniform peer samples if we filter out both the known root nodes and the node producing the peer samples.

8 Reference implementation

Included in Appendix B is a simple reference implementation in erlang, a high-level interpreted language. The simplicity of the algorithm is reflected in the fact that the entire implementation is under 200 lines of code, including a generic implementation of continuous-time Markov chains.

Our implementation uses the built-in erlang messaging system. Replacing this with raw UDP would take only a few more lines since the message format is so simple.

We set $\lambda = 1.00s^{-1}$ and $\mu = 0.01s^{-1}$ and performed two tests: the first with 10 nodes and 1 known root nodes and the second with 1000 nodes and 10 known root node. For each test we recorded the peer samples produced by a single peer and performed chi-squared tests for both uniform distribution and pairwise independence of the samples. The powers returned from the tests were:

```
10 nodes / 1 known root (3,000 samples)
  uniform distribution: 0.177e-01
  pairwise independence: 0.322e-01

1000 nodes / 10 known roots (400,000 samples)
  uniform distribution: 0.478e-04
  pairwise independence: 1.0
```

In order to perform the uniform distribution test it is recommended that we have at least ten times as many samples as there are nodes in the network and for the independence test at least ten times the square of the number of nodes. The failure for the last independence test can be attributed to the fact that we did not obtain the required number of samples.

With $\lambda = 0.1s^{-}1$ and $\mu = 0.001s^{-}1$ we were able to run 100,000 nodes plus 100 known root nodes on a single-core netbook. However, in this configuration gathering this number of samples would take longer than a year. Instead we pooled all the samples in the network to perform the tests.

```
100,000 nodes / 100 known roots (10,000,000 samples)
  uniform distribution: 0.00
  pairwise independence: 0.00
```

This gave us a large enough number of samples which were sufficiently well distributed that the power for these two tests fell within the bounds of numerical error.

9 Conclusion

We presented in this dissertation a gossip algorithm which is simple, easy to implement and resilient to random failure and churn. The peer sampling service produced by this algorithm generates independent, uniformly distributed peer samples at each node when in the steady-state. This analysis is backed up both by formal verification using the PRISM model checker and by experimental results on a single machine. As far as the author is aware this is the first gossip algorithm to make any guarantees about the behaviour of its peer sampling service.

This algorithm has so far only been tested on a small number of machines. Given the sensitive timings involved it would be prudent to test this on a large network to ensure that delays due to real life constraints do not damage the results.

We have proved that the samples at each node are independent and we have demonstrated in PRISM that the most recent samples for any pair of nodes are independent. It seems likely that all the peer samples across all nodes are independent. The proof of this, however, has been elusive because the peer samples taken together with their ordering in time are *not* independent (for example if node7 was the last node to produce a peer sample then the next sample is more likely to be 7). If we had chosen a notation that was more easily divorced from the timings this independence may have been easier to prove.

In unreliable networks, extra load is placed on the root nodes proportional to the rate of message failure. This is due to the naive error-handling strategy used in the current algorithm. Most gossip algorithms in large scale networks cache previously seen nodes and reuse these addresses in the case of failure. It would be useful to model the tradeoff between increased robustness and the uniform distribution of peer samples when using this strategy.

Large scale uses of gossip algorithms have often found that NAT negotiation is essential [10, 21] and that reliance on UPnP or similar is ineffective. The use in nthis algorithm of small and frequent messages to lots of different peers makes it difficult to perform NAT negotiation efficiently. A possible solution might be to reuse the root nodes as ICE servers for nodes which are known to be unable to open ports to the outside world.

This algorithm is suitable for use with trusted peers, for example within a data centre. Without further work it is not suitable for use with untrusted peers as it is susceptible to poisoning or other malicious behaviour. Future work could include identifying and managing malicious peers. In particular, it should be possible to rate-limit malicious peers without damaging the useful properties of the peer sampling service. This would go a long way towards preventing poisoning. Given the simplicity of this algorithm it may even be possible to establish bounds on the influence wielded by any given peer and thus show that specific class of attacks require an impractical amount of resources.

References

[1] Rena Bakhshi A and Maarten Van Steen A. An analytical model of information dissemination for a gossip-based protocol.

[2] André Allavena, Alan Demers, and John E. Hopcroft. Correctness of a gossip based membership protocol. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 292–301, New York, NY, USA, 2005. ACM.

[3] Lorenzo Alvisi, Jeroen Doumen, Rachid Guerraoui, Boris Koldehofe, Harry Li, Robbert van Renesse, and Gilles Tredan. How robust are gossip-based communication protocols? *SIGOPS Oper. Syst. Rev.*, 41(5):14–18, 2007.

[4] Rena Bakhshi, Francois Bonnet, Wan Fokkink, and Boudewijn Haverkort. Formal analysis techniques for gossiping protocols. *SIGOPS Oper. Syst. Rev.*, 41(5):28–36, 2007.

[5] Rena Bakhshi, Lucia Cloth, Wan Fokkink, and Boudewijn R. Haverkort. Meanfield analysis for the evaluation of gossip protocols. *SIGMETRICS Perform. Eval. Rev.*, 36(3):31–39, 2008.

[6] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Gossip algorithms: Design, analysis and applications. In *in Proceedings of IEEE INFOCOM*, pages 1653–1664, 2005.

[7] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Gossip algorithms: Design, analysis and applications. In *in Proceedings of IEEE INFOCOM*, pages 1653–1664, 2005.

[8] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107 – 117, 1998. Proceedings of the Seventh International World Wide Web Conference.

[9] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, New York, NY, USA, 1987. ACM.

[10] Niels Drost, Elth Ogston, Rob V. van Nieuwpoort, and Henri E. Bal. Arrg: real-world gossiping. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 147–158, New York, NY, USA, 2007. ACM.

[11] MARTIN ERWIG and STEVE KOLLMANSBERGER. Functional pearls: Probabilistic functional programming in haskell. *Journal of Functional Programming*, 16(01):21–34, 2006.

[12] Yaacov Fernandess, Antonio Fernández, and Maxime Monod. A generic theoretical framework for modeling gossip-based algorithms. *SIGOPS Oper. Syst. Rev.*, 41(5):19–27, 2007.

[13] Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. Purely functional lazy non-deterministic programming. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 11–22, New York, NY, USA, 2009. ACM.

[14] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3):8, 2007.

[15] Gian Paolo Jesi, David Hales, and Maarten van Steen. Identifying malicious peers before it's too late: A decentralized secure peer sampling service. In *SASO '07: Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems*, pages 237–246, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Gian Paolo Jesi and Alberto Montresor. Secure peer sampling service: the mosquito attack. In *Proceedings of the 5th WETICE Workshop on Collaborative Peer-to-Peer Systems (COPS'09)*, Groningen, The Netherlands, July 2009.

[17] Gian Paolo Jesi, Alberto Montresor, and Maarten van Steen. Secure peer sampling. *Computer Networks*, 2010. To appear.

[18] Anne-Marie Kermarrec and Maarten van Steen. Gossiping in distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(5):2–7, 2007.

[19] Marta Kwiatkowska, Gethin Norman, and David Parker. Analysis of a gossip protocol in prism. *SIGMETRICS Perform. Eval. Rev.*, 36(3):17–22, 2008.

[20] Jun Luo, Patrick Th. Eugster, and Jean-Pierre Hubaux. Route driven gossip: Probabilistic reliable multicast in ad hoc networks. In *IN PROC. OF INFOCOM*, pages 2229–2239, 2003.

[21] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, and H. J. Sips. Tribler: a social-based peer-to-peer system: Research articles. *Concurr. Comput. : Pract. Exper.*, 20(2):127–138, 2008.

[22] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, and H. J. Sips. Tribler: a social-based peer-to-peer system: Research articles. *Concurr. Comput. : Pract. Exper.*, 20(2):127–138, 2008.

[23] Xin Sun, Ruben Torres, and Sanjay Rao. Ddos attacks by subverting membership management in p2p systems. In *NPSEC '07: Proceedings of the 2007 3rd IEEE Workshop on Secure Network Protocols*, pages 1–6, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[25] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Middleware '98: Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 55–70, London, UK, 1998. Springer-Verlag.

[26] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13:197–217, 2005. 10.1007/s10922-005-4441-x.

Appendix A: Model checking methods

The PRISM models and properties in this dissertation are generated by a set of scripts in order to allow models with any number of nodes. A simple scripting harness was written for PRISM to allow automated testing. The default test suite tests each model with 3, 5, 10 and 20 nodes and is run as a post-commit on the darcs repository hosting this dissertation. Models with up to 100 nodes have also been tested in the same way but these test take too long to be run on every commit.

For models with up to 5 nodes the tests use model checking with steady-state properties. Each property is tested with every valid combination of nodes. For larger models the simulator is used. Each property is tested with only a few different nodes and we rely on the symmetry of the model to ensure that the property holds for all node. The simulator does not support testing steady-state properties so the same properties are expressed as instaneous rewards and tested at a range of times. For example, for the property:

```
S=? [ node0=0 ]
```

The matching reward is:

```
rewards "node0_0"
  node0=0 : 1;
endrewards
```

And we then test:

```
R{"node0_0"}=? [ I=T ]
```

Since the models converge quickly to the steady-state this has proved effective.

The simulator is run with a confidence of 0.5 and an approximation of 0.01. The results are tested for equality with a tolerance of 0.02 between the minimum and maximum value. We found in practice that the actual confidence is well within the specified bounds and none of these tests fell outside the tolerance.

The use of PRISM was essential in the experimental phases of this work. The ability to quickly test properties and iron out problems before attempting proofs saved much time and effort. This was invaluable for the problems in section 6 especially, where PRISM was used to detect subtle dependencies between peer samples at different nodes.

We encountered a number of minor difficulties when using PRISM. The main obstacle was the model description language, which has few mechanisms for abstraction. As a result we relied on a series of scripts to generate model descriptions for different network sizes. The length of the generated descriptions was quadratic in the network size which made checking the correctness difficult. The availability of arrays in the PRISM language would have greatly simplified the models in this dissertation. The lack of abstraction also makes it difficult to combine different modifications, such as churn and message failure, which would have been fairly trivial in a full programming language. There are a number of examples of embedding probabilistic description languages into existing programming languages [11, 13]. It could be worth investigating the feasibility of connecting one of these languages directly to the PRISM engine.

Another issue that arose was that the steady-state operator S=? can only handle binary properties when it seems it should be possible to handle any quantitive property. Instead, we tracked the past value of each variable explicitly in order to handle independence tests.

Finally, the generation of jump processes by hand is error-prone even for small models. PRISM could be extended to handle this translation automatically.

Appendix B: Reference implementation

The reference implementation contains two modules: ctmc.erl is a generic implementation of a CTMC and poppi.erl is a set of callbacks implementing the CTMC for a single node.

```
-module(ctmc).

-export([start/2, call/2]).
-export([behaviour_info/1]).

-behaviour(gen_server).
-export([code_change/3, handle_call/3, handle_cast/2,
    handle_info/2, init/1, terminate/2]).

-record(ctmc, {module, state, next_event}).
-define(SERVER, ?MODULE).

behaviour_info(callbacks) ->
    [{init,1},{events,1},{handle_event,2},{handle_call,2}];

behaviour_info(_Other) ->
    undefined.

% api

start(Module, Args) ->
    gen_server:start(?MODULE, [Module, Args], []).

call(Ctmc, Call) ->
    try
        {ok, gen_server:call(Ctmc, Call, 1000)}
    catch
        _:{timeout,_} -> timeout
    end.

% gen_server callbacks

init([Module, Args]) ->
    random:seed(now()),
    State = Module:init(Args),
    {Next_event, Timeout} = next_event(Module, State),
    {ok, #ctmc{module=Module, state=State, next_event=Next_event},
        Timeout}.

handle_call(Call, _From, #ctmc{module=Module, state=State}=Ctmc) ->
    {State2, Reply} = Module:handle_call(State, Call),
    {Next_event, Timeout} = next_event(Module, State2),
    {reply, Reply, Ctmc#ctmc{state=State2, next_event=Next_event},
        Timeout}.

handle_cast(_Cast, _Ctmc) ->
    ok.

handle_info(timeout, #ctmc{module=Module, state=State,
    next_event=Next_event}=Ctmc) ->
    State2 = Module:handle_event(State, Next_event),
    {Next_event2, Timeout} = next_event(Module, State2),
    {noreply, Ctmc#ctmc{state=State2, next_event=Next_event2},
        Timeout};

handle_info(_, #ctmc{module=Module, state=State}=Ctmc) ->
    {Next_event, Timeout} = next_event(Module, State),
    {noreply, Ctmc#ctmc{next_event=Next_event}, Timeout}.

terminate(_Reason, _State) ->
  ok.
```

```
code_change(_OldVsn, State, _Extra) ->
  {ok, State}.

% internal functions

next_event(Module, State) ->
    random:seed(now()),
    Events = [{Event, Rate} || {Event, Rate} <-
        Module:events(State), Rate > 0],
    Total = lists:sum([Rate || {_Event, Rate} <- Events]),
    case choose_event(Total * random:uniform(), Events) of
        no_event -> {no_event, infinity};
        {event, Event} -> {Event, round(1000 * exponential(Total))}
    end.

% rounding errors may occasionally cause spurious no_event
% normally should only happen when the event list is empty
choose_event(_P, []) ->
    no_event;
choose_event(P, [{Event, Rate}|Events]) ->
    New_p = P - Rate,
    if
        New_p =< 0 -> {event, Event};
        true -> choose_event(New_p, Events)
    end.

exponential(Lambda) ->
    P = random:uniform(),
    (-math:log(P) / Lambda).
```

```
-module(poppi).

-export([start/0, start/1, start_experiment/3]).

-behaviour(ctmc).
-export([init/1, events/1, handle_event/2, handle_call/2]).

-record(poppi, {node,root,known_roots,log}).

% event constants

-define(LAMBDA, 0.1).
-define(MU, 0.001).

% api

start() ->
    start([]).

start(KnownRoots) ->
    start(KnownRoots, fun (_) -> ok end).

start(KnownRoots, Log) ->
    Poppi = #poppi{node=none, root=none, known_roots=KnownRoots,
        log=Log},
    ctmc:start(?MODULE, [Poppi]).

start_experiment(N, K, Sampling) ->
    Log =
        case Sampling of
            one -> fun (_) -> ok end;
            all -> fun log_sample/1
        end,
    KnownRoots = [Root || {ok, Root} <- [start([], Log) || _ <-
        lists:seq(1,K)]],
    _Nodes = [Node || {ok, Node} <- [start(KnownRoots, Log) || _
        <- lists:seq(1,N-1)]],
    {ok, _ExperimentNode} = start(KnownRoots, fun log_sample/1).

% ctmc callbacks

init([#poppi{known_roots=KnownRoots}=Poppi]) ->
    Node = choose_root(KnownRoots),
    Root = choose_root(KnownRoots),
    Poppi#poppi{node=Node, root=Root}.

events(#poppi{}) ->
    lists:flatten([{msg_root, ?LAMBDA}, {msg_known_root, ?MU}]).

handle_event(#poppi{node=Node}=Poppi, msg_root) ->
    %log("Messaging root ~w from ~w~n", [Node, self()]),
    sample(Poppi, Node);

handle_event(#poppi{known_roots=KnownRoots}=Poppi, msg_known_root)
    ->
    Root= choose_root(KnownRoots),
    %log("Messaging known root ~w from ~w~n", [Root, self()]),
    sample(Poppi, Root).

handle_call(#poppi{root=Root}=Poppi, {sample, Node}) ->
    %log("Sending sample to ~w from ~w~n", [Node, self()]),
    {Poppi#poppi{root=Node}, Root}.
```

```
% internal functions

sample(#poppi{root=Root, known_roots=KnownRoots, log=Log}=Poppi,
    Target) ->
    if
        Target == self() ->
            Log(Root),
            Poppi#poppi{node=Root, root=self()};
        true ->
            case ctmc:call(Target, {sample, self()}) of
                {ok, Node2} ->
                    Log(Node2),
                    Poppi#poppi{node=Node2};
                timeout ->
                    log("Timeout at ~w~n", [self()]),
                    Node2 = choose_root(KnownRoots),
                    Log(Node2),
                    Poppi#poppi{node=Node2}
            end
    end.

log(Format, Args) ->
    io:format(Format, Args).

log_sample(Sample) ->
    {_,Seconds,_} = erlang:now(),
    log("~w: ~w~n", [Seconds, Sample]).

choice(List) ->
    case length(List) of
        0 -> none;
        N -> {choice, lists:nth(random:uniform(N),List)}
    end.

choose_root(Roots) ->
    case choice(Roots) of
        none -> self();
        {choice, Root} -> Root
    end.
```