RayNet Specification

April 4, 2008

This purpose of this document is to specify the behavior of the RayNet overlay []. Some basic design issues are also discussed, thus providing a starting point for the implementation of RayNet in a real-world setting. The document is structured as follows. In Section **??** we present the general skeleton of gossip-based view maintenance protocols. In Section 1 we present two such protocols that constitute the basis of Raynet, and finally in Section **??** we present the architecture of Raynet and discuss the details of its operation.

1 Gossip-Based View Maintenance

The behavior of RayNet follows the general scheme of a Gossip-Based overlay maintenance service. Specifically, Raynet is based on the combination of two gossip-based view-maintenance components that contribute to building its fault-tolerant proximity-aware overlay structure. In the following, we describe the general protocol framework that lies at the basis of each of these view maintenance protocols.

A gossip-based view maintenance protocol, also referred to as peer-sampling service, maintains an unstructured overlay network using a gossiping procedure. Each peer maintains a data structure consisting of a list of other peers, its neighbors. This list, normally referred to as *view*, is constantly refreshed using information exchanged with other nodes in the overlay.

As described in [], the skeleton of gossip-based peer sampling protocols conforms to a simple general model and mainly differ in the way in which the view is updated with each gossip exchange. The rest of this section provides a description of this framework, while Sections 1 and **??** instantiate it into the building blocks of RayNet.

Peer-Sampling Framework In general, each network node is associated with an address (e.g. IP and port) that is needed for sending a message to that node. In addition it may also be associated with information needed in specific peer-sampling instances. Each node maintains a membership table, *view*, representing its knowledge of the global membership. This view is a list of c *node descriptors*. Parameter c represents the size of the list and is the same for all nodes.

A node descriptor contains the information associated to the node (as defined above) and an *age value* that represents the freshness of the given node descriptor. The partial view is a list data structure, and accordingly, the usual list operations is defined on it. The protocol also ensures that there is at most one descriptor for the same node in every view.

The purpose of the gossiping algorithm, executed periodically on each node and resulting in two peers exchanging their membership information, is to update the partial views to reflect the dynamics of the system. Each node executes the same protocol, of which the skeleton is shown in Algorithm 1. The protocol responds to two events: a periodic timer event triggers the execution of the active cycle initiating communication with other nodes, while the receipt of a message invokes the passive cycle responsible for answering other nodes' requests.

Active Cycle The active cycle is activated exactly once every T time units. First, it selects random node from the view to exchange membership information with.

Second, the peer executing the active cycle exchanges membership information with the selected peer by generating a gossip message. The content of this message is determined by the selectToSend(recipient) method. Choosing a specific implementation of this method, allows each protocol instance to determine the which peers to include in propagated messages, ultimately controlling the characteristics of the resulting overlay.

Finally, at the end of the cycle, the age of each peer in the local view is incremented by one.

Passive Cycle The passive cycle reacts to the receipt of a message from another peer. The message may either be a spontaneous message (i.e. sent in the active cycle), or a response (sent in the passive cycle as described in the following). In the first case, the protocol generates a response message whose content is again determined by invoking the selectToSend(recipient) method.

After sending the message, or after determining that the received message was already a response, the protocol proceeds to updating the local view using the another protocol-specific method, selectToKeep(buffer, lastSent). The method takes as parameters the list of node descriptors received with the gossip message being processed, as well as the list of descriptors sent in the last gossip message to the peer whose message is being processed. The first parameter allows it to update the view with the new nodes; the second allows the update mechanism to take into account the information exchanged with the other peer. When removing descriptors from the view, it is in fact beneficial to remove those that have been sent to another peer, as this minimizes the overall loss of information.

Finally, as in the case of the active cycle, the age of each peer in the local view is incremented by one.

```
global variables:
set[node] lastSent

activeCycle  begin
    peer = select random peer from view
    message = create new message
    message.buffer = selectToSend(peer)
    m.sender=self
    lastSent=message.buffer
    send message to peer
end

passiveCycle(Message m)  begin
    if not m is Response then
        response = create new Response
        response.sender=self
        response.buffer = selectToSend(peer)
        lastSent=response.buffer
        send response to m.sender
    selectToKeep(m.buffer, lastSent)
end

main active thread  begin
    while true do
        activeCycle()
        sleep(gossipInterval)
end

main passive thread  begin
    while true do
        receive (message, sender)
        passiveCycle(message, sender)
end
```

Algorithm 1: Skeleton of a gossip-based view-maintenance protocol

2 Instantiating the Peer Sampling

The generic protocol described above is characterized by three properties: namely, the selection of the peer to gossip with, the selection of the information to be gossiped, and the computation of the new membership table based on the exchanged information. In the following, we present the two instances of the generic protocol that constitute the raynet overlay.

2.1 Kleinberg-Like Link Maintenance

The first instance we consider is a mechanism to sample the nodes in the network, so that each peer obtains a set of neighbors such that the resulting overlay has the characteristics of a small world network. This is, in fact, at the basis of RayNet's ability to route to any node in space in polylogarithmic time. In the following, we present the protocol by describing the instantiation of the two generic methods in the protocol framework.

selectToSend The selectToSend(recipient) operation is designed to return a set of peers that are not immediately useful for the local peer's view. This is convenient as these peers will be later removed by the selectToKeep call. To achieve this biased peer selection, the method first computes a temporary view V' by removing the peer selected by selectPeer (the destination of the gossip message) from the local view.

Then each peer p in V' is associated with a probability of being chosen $P_c(p)$ that depends on its distance from the local node. In the context of *Solipsis* this is effectively the cartesian distance between the locations of the site nodes. The probability follows a harmonic distribution and is computed with the following formula where $d(n,p)$ denotes the distance between the local node and the peer being considered.

$$P_c(p) = \frac{\frac{1}{d(n,p)^2}}{\sum_{p \in V'} \frac{1}{d(n,p)^2}}$$

After associating these probability with the peers in the view, the function uses them to extract a set K of $c/2$ peers that are of interest to the current node. The remaining peers, that is $V' \setminus K$ are then returned to the generic protocol.

selectToKeep The method selectToKeep(buffer, lastSent) modifies the local view based on the received buffer with the objective to keep the most interesting peer for the local node. To achieve this, it first appends the received buffer to the view. Then it eliminates duplicate entries from the view, and finally it removes items from the view starting with the items contained in lastSent until the size of the view is equal to c.

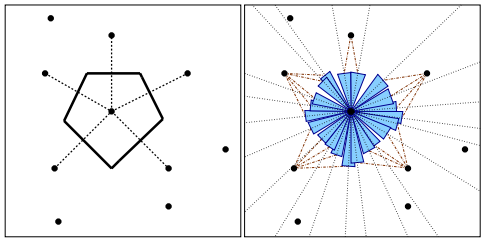

Figure 1: Volume estimation using rays (o is the central point corresponding to the local node).

2.2 RayNet Architecture

The RayNet protocol can also be described as a gossip-based view maintenance protocol. However, during its operation, it also exploits information contained in the views of the kleinberg peer sampling protocol described above. In the following we will refer the kleinberg view as V_K, while the RayNet view consisting of the voronoy neighbors will instead be referred to as V_V In order to guarantee successful routing, the size of V_V is set equal to $c = 10$.

2.2.1 selectToSend

The method returns a random subset of V_V containing $c/2$ nodes and not containing the node selected by select peer.

2.2.2 selectToKeep

The selectToKeep operation is the core of the RayNet protocol. Different, from the previous protocol, in RayNet this method evaluates the goodness of a view as whole, rather than examining each peer individually.

The protocol operates by constructing a set RAYS of R *rays* whose starting point is the local peer and whose and directions are drawn uniformly at random on the unit hyper-sphere. This is achieved using the method described in [**?**] that provides uniform probability distribution of points on the hyper-sphere. More precisely, the function CREATERAYS(R) generates R points uniformly distributed on the surface of the sphere centered at the local node, according to virtual world coordinates. Each ray is then univocally determined by the one point in the sphere and the local node.

Rays (dashed lines starting from o on Figure 1) will act as *probes*, to discover the closest intersection point p_{int} lying on the ray r with a (virtual) Voronoï cell of another node in the configuration, this node being the node n' for which $\lambda = d(p_{int}, n) = d(p_{int}, n')$, where n is the local node, is minimal. This allows Algorithm 2 to compute the estimated value of the voronoi cell associated with a set of nodes. Specifically, in the algorithm, the function compDistOnRay() in Algorithm 2 simply computes, for a pair consisting of a ray and a node n, the

point on the ray such that $\lambda = d(p_{int}, n) = d(p_{int}, n')$ and returns the value of λ.

```
    the set rays is essentially a set of points: set[Point] where Point x,
    represents a point x in the virtual world
    calcVolume(set[nodes] config)
    begin
      DOUBLE Λ[] ← ∅
      RAYS ← createRays(R)
(a)   for r ∈ rays do
        double λ ← ∞
        for node n_j ∈ config do
          double l ← compDistOnRay(r,n_j)
          if l < dist then
            λ ← l
(b)     Λ ← Λ · λ
      /* BallVol contains the unit Ball volume in dimension d */
(c)   return (BallVol × Σ_{λ∈Λ}(λ^d)) / R
    end
```

Algorithm 2: Monte-Carlo algorithm for estimating the volume of the cell for node n .

Lines (a) to (b) of Algorithm 2 present the selection of the closest peers for each ray. The function keeps all λ values for each ray (set Λ), and uses them to compute the estimation of the cell volume as follows (line (c) of Algorithm 2). Each ray r is associated to a *ball* of radius λ_r whose volume is given by $(BallVol \times (\lambda_r)^d)/R$, where $BallVol$ is the volume of the unit ball in dimension d. The volume of the estimated cell is the average value, for all rays, of volumes of such balls (the contribution for each ray is represented as grey cones on Figure 1). Such an estimator of the volume of the Voronoï cell is clearly unbiased, so that the estimated volume converges to the volume of the Voronoï cell when $R \to +\infty$. Nevertheless, the convergence strongly depends on the shape of the Voronoï cell, thus imposing the use of a large enough number of rays (10^3).

Computing the best configuration The algorithm to compute the best configuration is shown in Algorithm 3. The algorithm maintains a bipartite graph *best* containing on one side the nodes in V_V, and on the other side the rays R. We denote by $best_n(r)$ the Voronoï neighbour n_r of n according to ray r: it is the node n_r such that a ray issued from n and whose direction is r first reaches the Voronoï cell of n_r (this entry is never empty). Similarly, we denote by $\{best_R(n^*)\}$ the set of rays for which n^* is the current Voronoï neighbour of n (this set may be empty).

The objective is as follows: to compute the new view, for each node n_j in $buffer \setminus V_V$ (*i.e.* all peers for which $\{best_R(n_j)\}$ does not contain any infor-

mation), we determine the set of rays for which n_j is the Voronoï neighbour of the local node, n, in the augmented view Voronoï diagram. This operation is described by lines (a) to (b) of Algorithm 3. Peers found to be a Voronoï neighbour of n for a given ray are stored in the set *improve*, which has the same semantic as *best*, except that entries for some rays can be empty.

On line (c), either *improve* or *best* has information, for each ray, about which peer in the augmented view is a Voronoï neighbour of n. The next step is to compute to which extent each peer is needed in the new configuration. More precisely, given a peer n_x, we compute the volume of the cell of n with all peers *but* n_x (lines (c)-(d)). If the volume of the cell increases dramatically, that means that peer n_x was mandatory to ensure closeness and proximity. On the other hand, if the volume remains the same, then peer n_x has no contribution to coverage nor closeness.

Volumes associated to each peer (*i.e.* the volume without that peer in the configuration) are stored in the map *volumes*. This map is then sorted by decreasing volume values : starting from entries of peers that contributes highly to coverage and closeness, to entries of peers that have no or few contribution to coverage and closeness. The new configuration is built from the c peers that presents the maximum contribution, *i.e.* peers of the first c entries of *volume*.

3 Initialization

An important point when dealing with view maintenance protocols like raynet, is the initialization of their views. In the case of RayNet this process requires the insertion of a set of descriptors of known nodes (at least one) in both the Kleinberg and the RayNet view, that is in both V_K and V_V. Both views may be initialized with the same set.

4 Routing in the RayNet

The interface operation supported by the RayNet is ROUTE(MESSAGE M, POINT P) that routes a message to the node whose voronoi cell contains point p. This is achieved by the simple steps shown in Algorithm 4.

selectToKeep(set[nodes] $buffer$, set[nodes] *lastSent*)
/* lastSent is not used in this protocol */
Local variables:
improve (map ray → node) init ∅ /*improve has the same semantic as $best_n$*/
volumes (list of pairs (node,volume)) init ∅
begin
(a) **foreach** *ray* $r \in o.rays$ **do**
double $best_\lambda = \perp$
node $imp = \perp$
foreach *node* $n_j \in (buffer \cup V_K \setminus V_V)$ **do**
$\lambda \leftarrow \text{distOnRay}(r, n_j)$
if $\lambda < \begin{cases} best_n(r) & \text{if} \quad best_\lambda = \perp \\ best_\lambda & \text{if} \quad best_\lambda \neq \perp \end{cases}$ **then**
$imp \leftarrow n_j$
$best_\lambda = \lambda$
if $best_\lambda \neq \perp$ **then**
(b) improve[r] = imp
(c) **foreach** *node* $n_x \in V_V \cup (buffer \cup V_K \setminus V_V)$ **do**
(d) volumes ← volumes ∪ pair(n_x,calcVolumeImproved($best \cup improve, (buffer \cup V_K \setminus V_V) \setminus \{n_x\}$))
sort volumes by decreasing volume
$V_V \leftarrow \{volumes[1], \ldots, volumes[c]\}$
update $best_n$ and $best_R$
end

Algorithm 3: Update of node's view V_V. Sets $best_N$ and $best_R$ are constructed and coherent i.r.t. the current V_V when starting the algorithm.

```
route(Message m)
begin
    double d=∞
    node closest=null
    Point p=m.dest
    foreach node n_j ∈ V_K ∪ V_V ∪ {n} do
        if d > d(n_j, p) then
            d=d(n_j, p)
            closest=n_j
    if closest=n then
        return m to application
    else
        send m to closest
end

receive(Message m)
begin
    route (m)
end
```

Algorithm 4: Routing a message to a point.