1 Definitions

Solipsis Objects The *Solipsis* metaverse consists of a set of *entities*, each belonging to one of three categories: *avatars*, *objects*, and *sites*. Avatars are the main actors of the metaverse as they are the only entities that are capable of autonomous movement. In most cases they are virtual representations of the users of the metaverse and are directly controlled by them using the navigator platform. Objects, on the other hand, are virtual representation of entities from the real world such as furniture, books and whatever object may be moved or picked up by an avatar. Avatar and objects may also be robotic-like entities controlled by user-defined software components. Finally, sites constitute the basic building blocks of the metaverse and represents portions of the virtual space that may be occupied by objects or in which avatars can roam.

Avatars, objects, and sites are all associated with 3D-descriptions, which constitute the basis for the rendering of entities in the navigator platform. The 3D description of an entity contains all the information necessary for its 3D visualization: this includes a mesh- or prims-based model, a set of textures, and, in the case of avatars or objects, an animation. In addition the 3D-description may contain sounds, videos, or multimedia content associated with the entity. From the point of view of this protocol, such a representation merely consists of a set of binary objects, or files, that should be provided to the rendering component. Each such object is associated with its own version number and may thus be updated independently of the others.

The state of an entity in the metaverse is determined by an *entity descriptor*, from now on referred to simply as *descriptor*. An entity's descriptor consist of the minimal information required to display a rough representation of the entity and includes, for example, the its location, its shape, its size, and its orientation.

Descriptors also keep track of which hosts have recorded a copy of the corresponding entity's 3D description. Specifically, each descriptor contains two fields associated with each of the files that constitute the a 3D description. The first field is the latest known version number of the file, while the second is a list of hosts that are known to have cached a copy of the version of the file indicated in the first field.

The descriptor is the means through which the *Solipsis* protocol tracks the evolution of an entity. To maintain consistency, each descriptor is associated with a sequence number that is incremented each time the descriptor is modified. A minimal descriptor is shown in Table **??**.

1.1 Solipsis Hosts and Nodes

From a practical viewpoint, the *Solipsis* platform is distributed over a set of *hosts*, also referred to as *peers* that maintain information about every entity that is currently present in the metaverse. Each host is associated with a single instance of the *Solipsis* platform, and throughout its lifetime, it may create new entities or destroy previously created ones according to the requests of the navigator component. Also, similar to an entity, each host is associated with a

UID	universal identifier of the entity
seqNum	sequence number
owner	identifier of the node managing the entity
type	site or avatar
loc	location in the 3D space
ori	orientation in the 3D space
shape	shape from a predefined set
box	bounding box of the object
R_p	perceptibility radius: distance from which the object is visible in the absence of obstacles
R_b	radius of smallest sphere enclosing the entity
objs_a	list of entities attached to the current one
f_1	first file of 3d-description
v_1	version number for first file
c_1	list of hosts that have cached v_1 of f_1
...	...
f_n	n-th file of 3d-description
v_n	version number for n-th file
c_n	list of hosts that have cached v_n of f_n
...	additional fields for progressive levels of details

Table 1: Format of an object descriptor

unique identifier (UID).

Because each host may be responsible for several entities at the same time, we define a *(Solipsis) node* as the set of resources dedicated to the management of a given entity. Each node incapsulates information about the corresponding entity's descriptors as well as the threads of control responsible for managing all the interactions of the node with the rest of the *Solipsis* metaverse as well as with the navigator platform. Depending on the type of entity they are managing, we distinguish three types of *Solipsis* nodes: *site nodes*, *avatar nodes*, and *object nodes*.

1.2 P2P Architecture

Nodes are organized in a three-dimensional overlay network based on the Raynet protocol. The arrangement of nodes in this overlay is based on the locations of the corresponding sites in the *Solipsis* world. In particular, each site node is responsible for a section of the world, its *cell*, centered around its own site and comprising a region of space around it. This is achieved by subdividing space according to a Voronoi tessellation, as shown in Figure **??**. Neighborhood relationships between peers are determined by the distance between the corresponding points in this space. Specifically, two Raynet nodes are neighbors in the overlay if the two corresponding points are neighbors in the Delaunay graph comprising all the points associated with Raynet nodes.

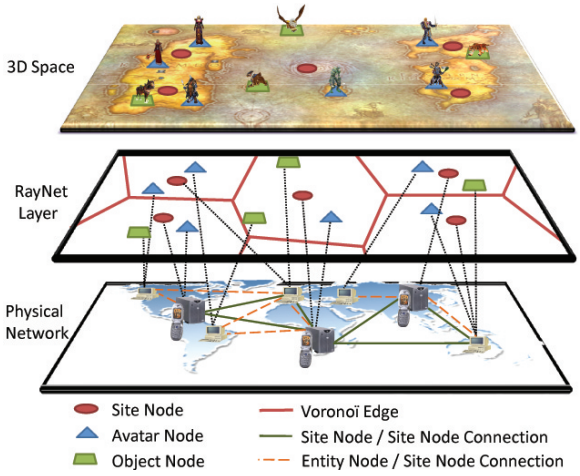

Figure 1: The p2p architecture showing the mapping of the raynet layer on the virtual world layer according to the proximity of nodes in the virtual space. The host layer shows clients connected into a peer to peer scheme.

Given a set of generator points$\{p \in \Re^d\}$, the Delaunay graph is obtained as the adjacency graph of the corresponding Voronoi diagram, which in turn is a tessellation of the d-dimensional space $\Re^d$. Specifically, the cell in the tessellation associated with a point p_x is such that it contains all the points that are closer to p_x than to any other generator point in the set.

2 Solipsis Protocol Details

2.1 Constants

The following is a list of constants that affect the protocol operation.

SAFED	distance between the bounding spheres of objects below which the corresponding nodes must update their positions at a high frequency.
NEIGHD	distance within which objects should exchange position updates lazily, resulting in a lower update frequency.

2.2 Data Structures Maintained at Node

Solipsis nodes maintain a set of data structures to manage the interaction with the objects that constitute the 3D world. In the following, we describe each of these structures for site nodes and for avatar nodes, respectively.

vLIST
maintained by: Site Node

Each site node maintains a visible list, vLIST. The vLIST is a table containing an entry for each object that is visible from the current site node's cell. Each entry in the list consists of an object descriptor and of a timestamp indicating when the entry was last updated. Each object in the table may represent a site or an avatar and may be inside or outside of the current node's cell. However, the implementation may use separate tables for each of these sets of object, as long as the protocol's semantics remain the same as specified in this document.

vLIST
maintained by: Avatar Node

Similar to site nodes, avatar nodes also maintain a visible list, vLIST, containing an entry for each object that is potentially visible from the avatar's location. As in the case of site nodes, each object in the table may represent a site or an avatar in any of the surrounding cells, although the implementation may also use a separate table for each set of objects.

sNODE
maintained by: Avatar Node

Each avatar maintains an sNODE pointer, that always refer to site node that is currently responsible for the avatar's location.

3DMODELS
maintained by: Host

Each host maintains a model table 3DMODELS that is accessible by all avatar and site nodes associated with the host. The table contains references to local copies of 3D models cached by the current host. At each instant the 3DMODELS table must always contain references to the 3D model of sites and avatars permanently associated with the hosts. In addition, it may contain references to remote avatars and sites as required for visualization purposes or in order to achieve persistency in the presence of failures.

2.3 RayNet

Both avatar and site nodes have access to a rNET object that abstracts their interface with the RayNet overlay network. The rNET object provides each site node with with the following two functions.

- GETNEIGHS(): may be called only by site nodes and returns the list of the node's neighbors.
- ROUTE(m, p): may be called by both avatars and site nodes and routes a

message m to the site node associated with the location p in the coordinate space.

2.4 Protocol Messages

Protocol Messages are exchanged asynchronously between Solipsis nodes using UDP. This may be overridden when the two communicating nodes (e.g. a site and an avatar) actually reside on the same peer.

2.4.1 View synchronization between avatar nodes

avUpdate
received by: avatar node
sent by: avatar node
w **when:** when moving, periodically with interval T_{upd}
parameters: ObjDescriptor nd
parameters: Set¡Descriptor¿ descA, Set¡Descriptor¿ descB

Avatar update messages are exchanged between neighboring avatars to maintain up-to-date information about close objects and particularly about those that may interact with their movements. Each avatar node should send an avUpdate message whenever it changes its position or other attributes such as parameters of its current animation. Moreover, it should send additional avUpdate messages to prevent their frequency from falling below one update every T_{upd} seconds. The contents of each avUpdate message consists simply of the object descriptor of the sending avatar.

Each avatar node computes the set of recipients of avatar update messages based on its size and that of the avatar nodes in its cell as specified in the node descriptors. Specifically, the set of recipients includes all the avatar that are at a distance that is less than or equal to the sum of their $R_b$values, plus the sender avatar's R_b, plus a safe distance parameter safeD. This guarantees that each update reaches all the avatars that may potentially collide with the current avatar before the next update[1].

Each avUpdate message comprises two sets of descriptors. The first always contains the node descriptor of the avatar sending and of any object that is directly or indirectly attached to the avatar. Information about the sending avatar and attached objects contributes to implement the first level of the update mechanism described in Section **??**. Specifically, it allows each avatar node to obtain accurate information about objects that may potentially collide with it in the near future. The second part of the avUpdate message is instead a list of node descriptors taken from the sending avatar's vList. More precisely, this list includes information about avatars (and attached objects) that are currently closer to the sending avatar than to the recipient of the message and that are at a distance that is less than neighD from the latter. Note that each avUpdate

[1] During standard operation

message does not need to contain all the node descriptors for all the objects that satisfy the above condition. On the other hand, each message only needs to contain information about a randomly chosen subset of such objects. [2] Still, if a message contains information about an avatar, then it must also contain information about all of the objects that are currently attached to it. This second part of the update message, effectively implements the second level of the object update mechanism described in section **??**.

An avatar node reacts to the receipt of an AVUPDATE by updating its visible list. Specifically, it inserts or updates the entry for the object descriptor listed in the message. If the VLIST contains no entry for a given object, then corresponding descriptor from the message is simply inserted in it. If, on the other hand, the VLIST already contains entry for the object, the receiving node replaces it with the received one only if the latter is associated with a fresher sequence number.

2.4.2 View synchronization between site and avatar/object nodes and between site nodes

AVNODEHBEAT
received by: site node
parameters: ObjDescriptor nd

Each avatar periodically sends an avatar heartbeat message to the site node responsible for the avatar's current location, SNODE. The interval between messages, T_{hb} should be on the order of seconds. However, avatar nodes may occasionally send more frequent updates, if their position or other features have undergone significant changes. The receiving site node reacts to the an AVNODEHBEAT message by updating its VLIST. Specifically, the entry associated with the avatar is updated either if it was absent from the VLIST or if the received entry has a fresher sequence number than the one in the table.

VUPDATE
received by: site node, avatar node
parameters: list<ObjDescriptor> nds

A view update message contains the necessary information to update the visible lists neighboring site nodes. The message is sent every T_u seconds by every site node to each of its neighboring site nodes (RNET.GETNEIGHS()). The contents of the message consist of a list of node descriptors representing all the objects from the sender node's visible list that are also visible from the sender node's current position. A site node that receives a VUPDATE message from one of its neighbors in the raynet overlay updates its visible list with all the node descriptors in the message.

[2] **Davide:** To get better guarantees, we need a raynet-like structures for avatars as well.

siteNodeHBeat
received by: avatar node
parameters: list<ObjDescriptor> nds

A view update message contains the necessary information to update the visible lists of the avatar nodes in a site node's cell. Similar to vUpdate messages, siteNodeHBeat messages are sentevery T_u seconds by each site node to all of the avatars that are currently in their cells. In this first version of the protocol sending should be done directly to all avatars in the cell. Future versions will include a more efficient data distribution strategy to reduce the forwarding load on site nodes.

Different from the case of vUpdate, the contents of the a siteNodeHBeat message comprise all the node descriptors of the objects that are in the sender site node's visible list. An avatar receiving the message, uses it to update its visible list. Specifically it adds to it all node descriptors in nds which are not already present. In addition, it updates each entry in the list for which there is a newer node descriptor (with a fresher sequence number) in nds.

It should be noted that T_u may be on the order of seconds or tens of seconds.

2.4.3 Managing Object Nodes

objRec
sent by: avatar node
received by: avatar node, site node, object node
when: when requesting management of object
parameters: UID objUID

The message is sent by an avatar node whenever it detects that it has collided with an object, or whenever the avatar intends to touch or pick up the object. After the message has been sent the avatar node may optimistically start managing the object but management should be continued only if permission is granted by the previous manager. Moreover, no updates about the object may be sent before permission is granted. If no response to the message is received within T_{acq}, the node assumes that the request has been lost and sends a new request.

A node receiving an objRecmessage regarding a given object, with identifier objUID, looks for the object's descriptor in its vList and verifies that its magNode field is equal either to its own UID or to that of the sender of the objRecmessage.[3] [4] If both constraints are satisfied, the receiving node updates the magNode field in the descriptor to match the UID of the sender of the objRec, and then responds with an objAckmessage. Otherwise, if either of the above conditions is not satisfied, the node replies by sending an objRefmessage.

[3] The latter condition is necessary to tolerate a certain degree of message loss.
[4] **Davide:** Add constraint for free/personal objects

It should be noted that according to the above description, if new requests are received for the same OBJUID from the same REQNODE, they are processed normally to provide some form of fault tolerance.

OBJACK
sent by: avatar node, site node, object node
received by: avatar node
when: when granting management of object
parameters: Descriptor objDesc

The message is sent by an avatar node, site node, or object node in response to an OBJRECin order to grant permission to manage the specified object.

Upon receipt of this message, the destination avatar node updates the corresponding descriptor in its VLIST and starts managing physics for the object.

Special processing carried out by object nodes After sending the message, an object node also starts a timer T_{del} (OBJUID), where OBJUID =OBJDESC.UID. If a previous timer was scheduled for the same OBJUID, then the node first cancels the previous timer and then starts a new one. At the expiration of the timer, the object node commits suicide and releases all resources.

OBJREF
sent by: avatar node, site node, object node
received by: avatar node
when: when refusing ownership of object
parameters: UID OBJUID, UID MANAGER

The message is sent by an avatar node, site node, or object node, to refuse a request to manage the object with identifier OBJUID. I a node refuses such a request because it is not the node managing the object, then it also records the identifier of the actual manager in the message.

An avatar node receiving the refusal message checks the value of the MANAGER field. If the value is not null then it sends a new request to the specified manager, otherwise it simply assumes that the management of the object cannot be acquired.

2.4.4 Joining Operations

Avatars An avatar node wishing to join the *Solipsis* needs two pieces of information: a joining position, and the identifier, or the IP and port, of at least one site node. The joining position should in general be the last of the avatar in the *Solipsis* world but may also be any other position, as in the case of an avatar joining for the first time. The identifier or IP address and port number of a site node are instead required to initialize the Raynet proxy so that it can effectively route messages enabling the avatar to join the *Solipsis* metaverse.

Once these two pieces of information are available, the actual joining operation consists simply of initializing the Raynet proxy and routing an avNodeHBeat message to the avatar's joining position.

Sites A site node wishing to join the *Solipsis* world[5] needs, at a minimum, two pieces of information, the desired position of its site and the identifier or IP address and port of an existing site node that is currently in the *Solipsis* world.

Once the two pieces of information are available, the site node initializes the RayNet layer by providing it with the information regarding an existing site node in *Solipsis*. Then after a setup time T_{setup}, it performs a join operation on the RayNet, thereby joining the overlay. From this moment on, the node starts listening for messages from avatars and neighboring site nodes and starts disseminating the update messages as described above.

2.4.5 Persistence

Persistence is the ability of the *Solipsis* metaverse to maintain information about an entity while the corresponding host is offline. This should be achieved both when hosts disconnect nicely, possibly informing other peers, and when they disconnect abruptly without any advance notice. In *Solipsis*, we consider two types of persistency: one regarding sites and one regarding objects. Avatars, on the other hand, disappear whenever the corresponding hosts disconnect and reappear upon its reconnection.

Site Persistence Site persistence is achieved in *Solipsis* through replication and the self-repairing capabilities of the Raynet overlay.

2.4.6 Managing 3D Descriptions

newCopy
received by: site node, avatar node
parameters: UID, ver, f, url

The newCopy message informs an avatar or a site node that a host has cached a new copy of ver version of the file f associated with the object identified by UID. The receiving node updates its private copy of the object descriptor to include a reference to the new cached copy. [6]

3Dreq
received by: host

Hosts may obtain copies of 3D models by contacting one or more nodes that have cached their latest version as specified in object descriptors. To achieve

[5] **Davide:** Here we need to add right management
[6] **Davide:** Need to agree on what to do with old versions

this, they should send a 3DREQ message specifying the identifier of the object for which they are retrieving the model, the type of model file they are retrieving (e.g. wireframe, animation, ...), and the desired version number. The message is addressed to hosts regardless of whether they host site or avatar nodes, and it triggers a response consisting of a 3DRESP message as specified in the following.

3DRESP

A host receiving a 3DREQ message from another host should react by looking up the desired 3D model in its own model table. If the table contains a version of the required 3D model that is at least as recent as the one requested by the 3DREQ, then it responds with a 3DRESP message that includes a copy of such version.

3 Pseudocode

1: **procedure** PREPAREHBRECIPIENTSprepares the list of recipients for site node heartbeats.
2: assert(SNHBREC is empty)
3: **for all** $d \in$ VLIST **do**
4: SNHBREC.insert(d.UID)
5: **end for**
6: $T_{hb} = T_u$/SNHBREC.size()
7: **end procedure**
8: **procedure** SENDSITENODEHBEATExecuted every T_{hb} seconds by each site node.
9: **if** SNHBREC is empty **then**
10: prepareHBRecipients()
11: **end if**
12: **if not** SNHBREC is empty **then**
13: desc = SNHBREC.first()
14: SNHBREC.erase(desc) SITENODEHBEAT.nds=VLIST
15: send SITENODEHBEAT to desc
16: **end if**
17: **end procedure**
18: **procedure** RECVSITENODEHBEAT(SITENODEHBEAT)
19: **for all** $d \in$ SITENODEHBEAT.nds **do**
20: old=VLIST.get(d.UID)
21: **if** $old = null \vee old.\text{SEQNUM} \leq d.\text{SEQNUM}+1$ **then** VLIST.put(d.UID,d)
22: **end if**
23: **end for**
24: **end procedure**
25: **procedure** SENDVUPDATEExecuted every T_u seconds by each site node.
26: VUPDATE.nds=VLIST
27: **for all** $n \in$ RNET.GETNEIGHS **do**

```
28:         send vUpdate to n
29:     end for
30: end procedure
31: procedure RecvVUpdate(vUpdate)
32:     for all d ∈ vUpdate.nds do
33:         old=vList.get(d.UID)
34:         if old = null∨old.seqNum ≤ d.seqNum+1 then vList.put(d.UID,d)
35:         end if
36:     end for
37: end procedure
38: procedure sendAvHeartBeatExecuted every T_u seconds by each avatar
    node.
39:     avNodeHBeat.nd=myDesc rNet.route (avNodeHBeat, myDesc.loc)
40: end procedure
41: procedure RecvAvHeartBeat(avNodeHBeat)
42:     old=vList.get(avNodeHBeat.nd.UID)
43:     if old = null ∨ old.seqNum ≤ avNodeHBeat.nd.seqNum + 1 then
    vList.put(avNodeHBeat.nd.UID,avNodeHBeat.nd)
44:     end if
45: end procedure
46: procedure sendAvUpdateExecuted every T_avu seconds by each avatar
    node.
47:     if avhbRec is empty then
48:         prepareAvUpdRecipients()
49:     end if
50:     if not avhbRec is empty then
51:         dest = avhbRec.first()
52:         avhbRec.erase(dest)
53:         avUpdate.nd=myDesc
54:         avUpdate.desc=getDescriptorsWithin(dest.loc neighD)
55:         send avUpdate to dest
56:     end if
57: end procedure
58: procedure RecvAvUpdate(avUpdate)
59:     for all d ∈ avUpdate.desc ∪ {avUpdate.nd} do
60:         old=vList.get(d.UID)
61:         if old = null ∨ old.seqNum ≤ d.nd.seqNum +1 then vList.put(d.UID,d)
62:         end if
63:     end for
64: end procedure
65: procedure prepareAvUpdRecipientsprepares the list of recipients for
    avatar update messages.
66:     assert(avhbRec is empty)
67:     for all d ∈ vList do
68:         if mayCollide(d) then
69:             avhbRec.insert(d.UID)
```

70: **end if**
71: **end for**$T_{\text{avu}} = T_{upd}$/AVHBREC.size()
72: **end procedure**
73: **function** MAYCOLLIDE(Descriptor d)determines if an object is within potential collision distance **return** MYDESC.loc.distance(d.loc) - MYDESC.R_b - d.R_b - SAFED ≤ 0
74: **end function**
75: **function** GETDESCRIPTORSWITHIN(Coord loc, double dist)returns descriptors within d meters
76: Set¡Descriptor¿ DESCS
77: **for all** $d \in$ VLIST **do**
78: **if** loc.distance(d.loc) $\leq$ dist **then** DESCS.insert(d)
79: **end if**
80: **end forreturn** DESCS
81: **end function**

procedure SENDOREQ(UID OBJUID)Executed when touching or picking up object.
Descriptor d=VLIST.get(OBJUID)
if not d is null **then**
OBJREC.OBJUID =OBJUID
send OBJRECto d.MAGNODE
end if
end procedure
procedure RECVOREQ(OBJREC)
Descriptor d=VLIST.get(OBJUID)
if old = null $\vee$ old.SEQNUM < AVNODEHBEAT.nd.SEQNUM **then** VLIST.put(AVNODEHBEAT.nd.UID,AV
end if
end procedure

TODO:
- think about whether we absolutely need a DHT and what the advantages and drawbacks of having it are.
- add assumptions section
- define identifiers
- incorporate signing mechanisms for important messages - define important - decide whether it is used - always - periodically - challenge-response
- update descriptor (or what else???) with following information about files: - hash of file - IP addresses of where to get it
- use DHT to get files when all else fails... (check)
- specify how to use DHT to map identifiers onto URL (make sure we need it)

4 Security