# Solipsis APIs and Protocols v.0.1

Gwendal Simon
gwendal.simon@rd.francetelecom.com

December 15, 2003

# Contents

**1 Introduction** **2**
1.1 Entity . . . 2
1.1.1 Constant Characteristics . . . 2
1.1.2 Variable Characteristics . . . 3
1.2 General Overview . . . 3
1.2.1 Node . . . 3
1.2.2 Navigator . . . 4
1.2.3 Command Interface . . . 4

**2 Node ↔ Node Communications** **5**
2.1 Initialization Messages . . . 5
2.1.1 Informations Retrieval . . . 5
2.1.2 Connection Algorithm . . . 5
2.2 Topology Preservation . . . 7
2.2.1 Disconnection Detection . . . 7
2.2.2 Neighbor Management . . . 7
2.2.3 Local Awareness Preservation . . . 8
2.2.4 Global Connectivity Preservation . . . 8
2.3 Service Notifications . . . 9

**3 Navigator ↔ Node Interface** **10**
3.1 Connection Procedure . . . 10
3.2 Procedures Callable by Navigator . . . 11
3.2.1 Quit . . . 11
3.2.2 Service Notifications . . . 11
3.2.3 Informations Retrieval . . . 11
3.2.4 Entity Control . . . 12
3.3 Procedures Callable by Node . . . 12
3.3.1 Self Informations . . . 12
3.3.2 Neighbor Informations . . . 12
3.3.3 Service Notifications . . . 13

# Chapter 1

# Introduction

A shared virtual world is a computer-generated space used as a metaphor for interactions. Entities, driven by users or by computer, enter and leave the world, move from one virtual place to another and interact in real-time.

The SOLIPSIS system is a network of peers, where peers are connected entities sharing a virtual space. It intends to be scalable to an unlimited number of users and accessible by any computer connected to the Internet. It does not make use of any server and is solely based on a network of peers.

In our system, each participating computer runs a specific software that holds and controls one or several peers. These peers implement the entities of the virtual world and "perceive" their surroundings. Peers can be lite pieces of software and SOLIPSIS aims to be accessible to low end computers at 56kbs and to mobile wireless devices and not only to full featured broadband connected engines. Connected peers may exchange data like video, audio, avatars movements or any kind of events affecting the representation of the virtual world.

## 1.1 Entity

An entity is a lite piece of software running by a connected computer. We describe in the following the applicative characteristics of an entity.

### 1.1.1 Constant Characteristics

These characteristics are defined at the node creation. They can not be modified.

**Identifier** Each entity is identified by a unique and constant string *id*. The identifier is generated by the address of the socket dedicated to SOLIPSIS: $(@IP, @PORT)$.

$$id = @IP : @PORT$$

**Pseudo** In real life, everyone is usually known through a name and not by an ID. In SOLIPSIS, each entity chooses a string that acts as pseudo. Note that there is no guarantee of pseudo uniqueness.

**Caliber** An entity is not only a point in the virtual world. Its shape is a disk of radius $ca$ centered on $pos_e$. The caliber $ca$ is a constant value impacting on the entity inertia. To avoid too large entities, we bound the value of $ca$:

$$ca \in [1 \ldots 1024]$$

### 1.1.2 Variable Characteristics

The following characteristics may be modified at any time.

**Position** Each entity determines its variable position in the Solipsis world. The world is a two-dimensional torus $T = \{(x, y) \in \mathbb{N}_{size_x} \times \mathbb{N}_{size_y}\}$ where $size_x$ and $size_y$ are the size of Solipsis world and $\mathbb{N}_k$ denotes the positive integers modulo $k$. We chose $size_x$ and $size_y$ very large: 128 bits each, so approximately $10^{75}$ different positions. So, entity position is:

$$pos \in [0 \dots 2^{128}] \times [0 \dots 2^{128}]$$

**Orientation** Each entity has a variable orientation *ori* determined by a positive integer:

$$ori \in \mathbb{N}$$

**Awareness Radius** The *Awareness Area* corresponds to the immediate surrounding of an entity. The distributed algorithms of Solipsis guarantee that each entity knows all entities within its awareness area. We define the awareness area of an entity $e$ by the disk of radius $ar_e$ centered on $pos_e$. As for human experience, the awareness radius is variable. For instance, a person has an area of interest of only several meters inside a crowded city and several hundreds of meters on a ocean. Each entity is responsible of adjusting its awareness radius:

$$ar \in [1 \dots 2^{127}]$$

This set of informations characterizes an entity in the world. When two entities $e_1$ and $e_2$ are connected in Solipsis, it means that $e_1$ knows all informations of $e_2$ and *vice versa*.

## 1.2 General Overview

In the Solipsis system, all of the entities have symmetric roles as in peer-to-peer system (or network of peers). Rather than being divided into clients and servers, a participant to a network of peers may act as both a client and a server. A key concept for peer-to-peer systems is to permit any two nodes to communicate with one another in such a way that either ought to be able to initiate the contact.

We claim that a network of peers could be a powerful tool for organizing a community sharing a virtual world. As such, each peer or node is an entity that participates to the topology maintaining for both its own interest and the common interest.

In one entity, we distinguish a part dedicated to the network of peers presence (Section 1.2.1) and a part for the Interface between an user and her entity (Section 1.2.2). Figure 1.1 is an illustration of the overall architecture.

### 1.2.1 Node

Each entity owns a module called *Node* assuming the charge to "be in the world", to participate to the network of peers and to collaborate to maintain the topology of Solipsis

This module is mandatory. It provides a set of algorithms allowing virtual world sharing by message exchanges with other entities. This module owns all informations on adjacent entities. It is the central module of the entity.

The communications between entities are described in Chapter 2. For more informations on the organization of Solipsis and the distributed algorithms of *Node*, please refer to [1, 2].

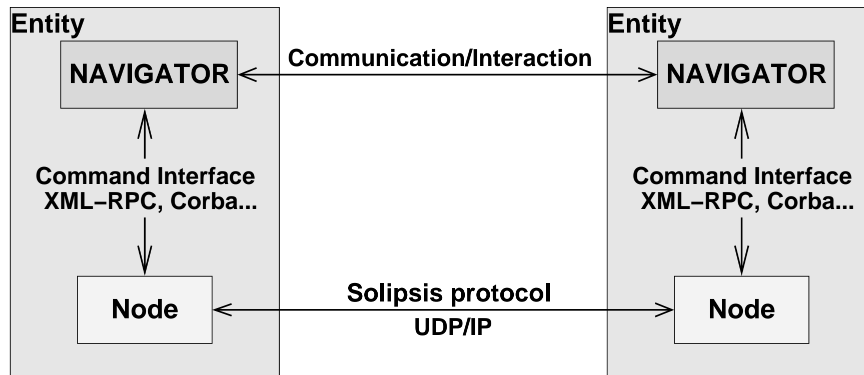


Figure 1.1: Overall architecture overview of SOLIPSIS.

### 1.2.2 Navigator

Communication between people in a virtual place is the *raison d'être* of the SOLIPSIS system. The *Node* module allows an entity to meet some other entities, but we need some additional modules for communications and interactions. These modules may also provide to user an illustration of the virtual world. We call them *Navigator* modules.

*Navigator* should act as a GUI between an user and an entity. It should offer to users the ability to stroll around and move from one virtual place to another.

Moreover, it should provide tools in order to communicate with other *Navigator*, e.g., by exchanging multimedia flows. Therefore, we define the notion of *Service*. A *Service* provide to a *Navigator* the ability to communicate using a peculiar media.

A *Service* is an independent part of a *Navigator* module. It owns an Internet address and it is able to communicate by a personal protocol with a similar service. For instance, we imagine a service for chatting, another one for file sharing, for avatar exchanges for 2D displayer, for 3D displayer or for videoconferencing.

The *Service* uses virtual informations kept by the *Navigator*. They are optional and may be discarded at any time.

### 1.2.3 Command Interface

There may be an infinity of such *Navigator* modules as a potential infinity ways of enjoying virtual experiences have good chance to raise. So, we choose to not merge *Navigator* modules and *Node* module in only one. Rather, we concentrate to only conceive a Command Interface between them. This Command Interface aims to allow:

- *Node* to inform *Navigator* on all events occurring in virtual surroundings,
- *Navigator* to control the entity by forcing *Node* to modify its variable characteristics.

*Navigator* and *Node* may be running on different computers and we let the opportunity to any new *Navigator* implementation to use SOLIPSIS. The Command Interface is described in Chapter 3.

# Chapter 2

# Node ↔ Node Communications

*Node* communication uses UDP protocol.

For each message description, we consider that an entity $e$ is the message sender and entity $e'$ is the message receiver. In following descriptions, text after item ← describes reasons that impose to $e$ to send the message. Text after item → are the consequences on $e'$ upon reception of message.

## 2.1 Initialization Messages

These messages help any entity to join the Solipsis system.

### 2.1.1 Informations Retrieval

The two first concerns of any joining entity are (i) how to find an entity connected to Solipsis, (ii) what is my network address. The following scheme resolves these two issues.

1. $e$ owns a list of entities potentially connected to Solipsis (entities it met during past experiences, downloaded list...). It sends to some of them a message indicating that it would connect to the world;

2. upon reception of a message of connection, a connected entity $e'$ sends to $e$ a response message containing $e$'s host and port it observes. By the way, it confirms its presence in Solipsis;

3. $e$ receives response from $e'$. It knows an entity connected ($e'$) and it retrieves accurate informations on its network identity.

### 2.1.2 Connection Algorithm

The Connection Algorithm consists in discovering the closest entity to the expected position, then detecting entities all around this position. It may be used not only at initialization of an immersion in Solipsis, but also when a connected entity decides to teleport.

#### Find The Nearest Entity

The entity $e$ only knows its identifier $id_e$ and its expected destination $pos_e$. It first has to meet the nearest entity to $pos_e$. The message $FindNearest$ contains informations on the querying entity $e$ and on its future position.

$$< FindNearest, host, port, pos_x, pos_y >$$

- ← entity $e$ sends this message either upon reception of $Nearest$ message, either at algorithm initialization.
- → entity $e'$ chooses the entity $e'' \in k(e')$ which is the nearest to the position. It answers by $Nearest$ containing informations on $e''$ or *Best* if it considers itself as the closest entity to the position.

A $Nearest$ message is the answer to a $FindNearest$ message containing informations on the nearest entity to the position carried by the $FindNearest$ message.

$$< Nearest, id, host, port, pos_x, pos_y >$$

- ← at reception of a $FindNearest$ message from $e'$, entity $e$ answers to $e'$ with characteristics of entity $e'' \in k(e)$, the nearest entity to the queried position.
- → with the entity $e''$ contained in the message, entity $e'$ knows that it keeps on discovering entites toward its future position. It is now able to send another $FindNearest$ message to $e''$. This continues recursively until $e'$ receives a *Best* message.

The *Best* message is another response to a $FindNearest$ message. It allows an entity $e$ to inform the requester that $e$ estimates that it is the nearest to the target.

$$< Best, id, host, port, pos_x, pos_y >$$

- ← at reception of a $FindNearest$ message from $e'$, entity $e$ answers with its own characteristics. By this way, it informs $e'$ that it considers itself as the nearest entity to the target.
- → when entity $e'$ receives a message *Best* containing informations on an entity $e''$, it knows the entity considered as the nearest to the target. It then sends a *QueryAround* message to $e''$.

### Query Entities All Around The Position

The entity $e$ knows an entity $e_c$ considered as the nearest to *pos* in SOLIPSIS. By now, it has to discover the nearest entities to *pos* all around its position. The *Query Around* message allows these discoveries.

$$< Queryaround, id, host, port, pos_x, pos_y, id^{e_c}, dist^{e_c} >$$

- ← an entity $e$ sends a *QueryAround* message when it considers that it detected the closest entity to the target, but it does not succeed in turning all around this position.
- → when entity $e'$ receives a message *QueryAround*, it first verifies whether it knows any neighbor whose distance to the target is less than $dist^{e_c}$. If it does know one, it sends a message $Nearest$ carrying informations on this neighbor. Otherwise, it chooses the neighbor that is the nearest to the target in the half-plane delimited by the line between itself and the target in the counter-clock wise. It then sends a message *Around* with informations on this neighbor to $e$.

A *Around* message is the answer to a *QueryAround* message. By the way, it means that entity $e'$ which received the *QueryAround* message from $e$ confirms the assumption that $e_c$ is the nearest entity to target. Moreover, entity $e'$ gives informations on an entity $e''$.

$$< Around, id, host, port, pos_x, pos_y >$$

- ← at reception of a *QueryAround* message, entity $e$ answers by a *Around* message if it does not know any entity $e''$ nearer to *pos* than $e_c$.

→ upon reception of a *Around* message from $e$, entity $e'$ verifies whether the turn is over. If the turn is over, $e'$ respects back the Global Connectivity at its new position. So, it connects with the collected entities. Otherwise, it sends a *QueryAround* message to the given entity.

## 2.2 Topology Preservation

Once the Connection Algorithm ends, an entity is connected to some entities within its awareness area. But, a shared virtual environment is dynamic and many events may impact on the topology: node motion, arrival and departure. Therefore, we need a mechanism to detect disconnecting entities (Section 2.2.1) and some messages to manage new connections (Section 2.2.2).

Moreover, these events may alter the topology of the network of peers. So, we provide a set of messages allowing each node to maintain locally the topology, expecting that the global topology could be preserved. We define two rules: the Local Awareness Rule and the Global Connectivity Rule [1, 2]. We describe in Section 2.2.3 and Section 2.2.4 the messages nodes use for rules preservation.

### 2.2.1 Disconnection Detection

Periodically, each entity notifies its aliveness by sending to all of its neighbors a message *Heartbeat*, carrying only its identifier *id*. Period between two consecutive *Heartbeat* emissions is universally fixed at 10 seconds.

$$< Heartbeat, id >$$

← every 10 seconds, entity $e$ sends *Heartbeat* messages to its neighbors.

→ upon reception of a *Heartbeat* message from entity $e$, entity $e'$ acknowledges the presence of $e$. If entity $e'$ does not receive any *Heartbeat* message from an entity $e$, it considers that $e$ leaved the system.

### 2.2.2 Neighbor Management

The message *Hello* allows an entity $e$ to open a connection with another entity $e'$. It contains informations on $e$:

$$< Hello, id, host, port, pos_x, pos_y, ar, ca, pseudo, ori >$$

← entity $e$ received previously from another entity some informations about an unknown entity $e'$. Trusting the entity that advises $e'$, entity $e$ decides to open a connection with $e'$. Mutual knowledge requires to provide to $e'$ the informations characterizing $e$.

→ entity $e'$ cannot refuse the connection and has to respond with a message *Connect* carrying its own informations.

The message *Connect* is the answer to a *Hello* message. It confirms the opening of a connection.

$$< Connect, id, host, port, pos_x, pos_y, ar, ca, pseudo, ori >$$

← entity $e$ received previously a *Hello* message. It accepts the connection and gives it back its owns informations.

→ entity $e'$ is the initiator of this connection. Naturally, it accepts it and updates the informations on $e$.

Sometimes, an entity has to close connections with a neighbor. It simply sends a message *Close* containing its identifier *id*.

$$< Close, id >$$

$\leftarrow$ an entity $e$ decides to drop out connections when its number of neighbors is too large. Entity $e$ cannot drop out connections with an entity that belongs to its Awareness Area (and respectively) nor with any entity $e'$ such that it bearks the Global Connectivity Rule. By default, entity $e$ chooses the most distant entity

$\rightarrow$ immediately upon reception of a *Close* message from $e$, entity $e'$ removes all informations on $e$.

### 2.2.3 Local Awareness Preservation

The *Local Awareness* property of an entity $e$ is ensured when $e$ is connected with all entities within its Awareness Area. Due to mobility, anytime, some entities may enter in its Awareness Area. We define in [2] a spontaneous collaboration scheme in which each entity sends a message when it detects that an entity enters in the Awareness Area of another.

This scheme requires that nodes are still informed on all events generated by their neighbors. Moreover, we provide a message for detection notification.

When an entity $e$ modifies one of its characteristics, it should inform its neighbors by sending a message *Delta* containing its identifier *id*, the type of the modified variable and the new value. If the modified characteristic is the position, then *var* is *POS* and the new value is the current position. Otherwise, *var* set at *AR* means that the awareness radius is now equal to $newValue$.

$$< Delta, id, var, newValue >$$

$\leftarrow$ when entity $e$ modifies one of its variables, it sends a *Delta* message to its neighbors.

$\rightarrow$ entity $e'$ should update its data and verify that these modifications have no topological impact.

A *Detect* message allows $e$ to provide to $e'$ characteristics of an entity $e'' \in k(e)$ with which $e'$ should be connected.

$$< Detect, id, host, port, pos_x, pos_y, ar, ca, pseudo >$$

$\leftarrow$ at reception of a *Hello* message or *Delta* message, entity $e$ detects that entity $e'$ and $e''$ should connect each other.

$\rightarrow$ entity $e'$ receiving a *Detect* message on a new entity $e''$ immediately opens a connection with $e''$.

### 2.2.4 Global Connectivity Preservation

If an entity does not know any entity in some large sector, it will hardly know about an entity arriving from this sector. Conversely, if it moves forward a sector with no known entity, it will hardly get aware of entities it should met on its path. The *Global Connectivity* property aims that an entity will not "turn its back" to a portion of the world [2].

A *Search* message allows $e$ to query to $e'$ an entity in the half-plane delimited by line $(e, e')$ in counterclockwise if $wise = 1$, clockwise else.

$$< Search, id, wise >$$

- ← we say that $e$ does not respect the Global Connectivity property when it exists two consecutive neighbors $e'$ and $e''$ such that $\angle(e' \ e \ e'') > \pi$. An entity that suddenly does not respect the Global Connectivity Rule must send a *Search* message in order to restore it. For an entity $e$, reception of *Delta* or *Close* messages from $e'$ or detection of $e'$ disconnection may imply a loss of Global Connectivity Rule respect. Moreover, when $e$ receives a *Found* message, it must verify whether the new entity is sufficient to restore the property.
- → entity $e'$ looks for an entity $e'' \in k(e')$ such that $e''$ is the nearest entity to $e$ locating in the half-plane.

The message *Found* is the answer of a *Search* message containing informations on an entity locating in the queried half-plane.

$$< Found, id, host, port, pos_x, pos_y, ar, ca, ori >$$

- ← at reception of a *Search* message, entity $e$ determines the closest entity to $e'$ in the queried half-plane. Entity $e$ does not send any message if it does not know any entity in the half-plane.
- → entity $e'$ should verify whether it respects back the Global Connectivity Rule.

## 2.3 Service Notifications

Each *Node* may be connected to a *Navigator* with some *Service*. The *Node* have knowledge of these *Service*. It should inform its neighbors on these communication modules.

$$< service, id, idService, descr, host, port >$$

- ← an entity $e$ sends such message when it receives from its *Navigator* a message notifying a new service. Additionaly, after the opening of every new connection, it sends too this message indicating to its new neighbor that it owns this service.
- → The entity $e'$ looks for self services that are compatible with this service. If there is one, it uses its Command Interface to give to its *Navigator* data allowing these services to initiate the communication.

$$< endService, id, idService >$$

- ← when a service module ends, it uses the Command Interface to inform *Node*. This information is sent by the entity $e$ to all of its neighbors.
- → The entity $e'$ looks for self services that are compatible with this service in order to inform them that the service is now closed.

# Chapter 3

# Navigator $\leftrightarrow$ Node Interface

We describe in the following the Command Interface between *Node* and *Navigator*. Note that these modules may be running on different computers on top of different languages. At first, they have to meet each other (Section 3.1). Then, we present the procedures on the *Node* remotely callable by any *Navigator* (Section 3.2). Lastly, we describe procedures on *Navigator* remotely callable by*Node* (Section 3.3).

## 3.1 Connection Procedure

We assume that the *Navigator* knows the host and the port of the *Node* to which it wants to connect. Moreover, we consider that the entity is alive and *Node* is listening any incoming TCP datagrams on this address.

Connection messages are:

$$< openGUI; protocol; host; port >$$

$$< openMedia; idMedia; protocol; host; port; pushOrPull >$$

The $openGUI$ message means that one user wishes to now control the entity. We do not authorize more than one GUI by *Node*.

The $openMedia$ message invites *Node* to provide to *Navigator* all informations about the vitrual world. Thus, the *Navigator* should be able to display a Solipsis experience. Note that we do not restrict the number of concurrent *Navigators* by *Node*. On the other hand, we do not authorize *Navigator* connection without previous reception of an $openGUI$.

The second parameter is the protocol used for the communications between *Node* and *Navigator*. By now, the only implementations of the Command Interface bases on XML-RPC, a remote procedure calling protocol. It allows software running on different operating systems and, thus, *Navigator* are able to call several procedures on the node and *vice versa*.

The third and fourth parameters are respectively the host and the port of the *Navigator*. Lastly, the last parameter indicates whether the *Navigator* wants to be connected in push mode (every information received by the node is forwarded to the *Navigator*) or in pull mode (the *Navigator* gets the informations when it requests them).

To both messages, the node responds either:

$$< accept; port >$$

The connexion is accepted, the additional informations is the port dedicated to the communications between the module and the node.

$$< refuse >$$

The connexion is refused.

The remainder of the communications is done with respect to the specified protocol, between the two dedicated ports.

## 3.2 Procedures Callable by Navigator

### 3.2.1 Quit

When the *Navigator* closes, it informs the *Node* by :

$$closeMedia(idMedia)$$

If the user wants to drop definitely its control on the *Node*, it drops simultaneously all communicating modules connected to the *Node*. To do this, it calls the procedure:

$$closeGUI()$$

Lastly, the *Navigator* may remove the entities from the SOLIPSIS world. This procedure stops the *Node*.

$$kill()$$

### 3.2.2 Service Notifications

A *Navigator* is characterized by some services, corresponding to its ability to exchange peculiar types of informations (video, chat, audio). The *Navigator* should inform the node about its services by sending the message:

$$addService(idMedia, idService, descrService, host, port)$$

The first parameter indicates that the identifier of the *Navigator*, the second parameter is the identifier of the service, the third parameter is a description of the service and the two last parameters are network informations.

At any time, the *Navigator* may send a addService message. It may too indicates that a service is now closed thanks to the message:

$$closeService(idMedia, idService)$$

The first parameter is the identifier of the *Navigator*, while the second parameter is the identifier of the service.

### 3.2.3 Informations Retrieval

**Get Informations :** Upon the module is ready to communicate with the node, it initiates the communication by calling the following procedure:

$$getInfos(idMedia, ALL, 0)$$

With this message, the module requests all informations known by the node, especially informations about the node itself and informations about the actual neighbors. This message may be used at any time when the module is in pull mode.

This message has a generic form :

$$getInfos(idMedia, var, addVar)$$

The different forms are :

- *var* set at “ME” with $addVar = 0$ for requirement of *Node* characteristics.
- *var* set at “SERVICE” with the identifier of a service for requirement of all data on this service (neighbors having this service, network informations. . . )
- *var* set at “ADJACENT” with the identifier of neighbor for the requirement of informations on a neighbor.

**Aliveness :** A *Navigator* may want to know whether the *Node* is still alive. It is possible to remotely call the procedure:

$$isAlive()$$

### 3.2.4 Entity Control

A *Navigator* may be able to control the entity. It informs the *Node* that a variable characteristic is modified by :

$$modSelf(var, delta)$$

If $var$ is $POS$, $delta$ is a vector $dX, dY$ representing the motion. If it is $ORI$, then $delta$ is the new orientation.

The *Navigator* may decide to teleport the entity to a completely new position by calling:

$$jump(pos_x, pos_y)$$

In this case, $pos_x, pos_y$ represents the new position of the entity.

## 3.3 Procedures Callable by Node

The node is able to call a set of procedures on the *Navigator*. We describe them in the following.

### 3.3.1 Self Informations

$$init(id, pos_x, pos_y, ar, calibre, pseudo)$$

These are the informations about the node. The first parameter is its identifier, the second and third ones are its absolute positions in SOLIPSIS world, the parameter $ar$ corresponds to the awareness radius of the node, the parameters *calibre* and *pseudo* are, as their names suggest, the calibre and the pseudo of the node.

$$modSelf(id, POS, pos_x, pos_y)$$

The node signals to the module that its position changed. $pos_x$ and $pos_y$ are the variations of the position.

### 3.3.2 Neighbor Informations

$$newNode(id, pos_x, pos_y, calibre, pseudo)$$

This message provides usual informations about a new neighbor.

$$modNode(id, POS, pos_x, pos_y)$$

The neighbor id changes its position. Parameters $pos_x$ and $pos_y$ correspond to the variation of its position.

$$deadNode(id)$$

The node $id$ is not any more the neighbor of the node.

### 3.3.3 Service Notifications

$$service(id, idService, descService, host, port)$$

This message signals that the node $id$ owns a service $idService$ whose description is $descService$. It provides too the network informations of this service.

$$closeService(id, idService)$$

This message signals that the service $idService$ for the node $id$ has quit.

# Bibliography

[1] J. Keller and G. Simon. Toward a Peer-to-Peer Shared Virtual Reality. In *IEEE Workshop on Resource Sharing in Massively Distributed Systems (RESH 02)*, july 2002.

[2] J. Keller and G. Simon. Solipsis: a Massively Multi-Participant Virtual World. In *International Conference on Parallel and Distributed Techniques and Applications (PDPTA 03)*, june 2003.