PlugNode - Connecting program modules efficiently on a distance as well as locally

Purpose

A PlugNode has been designed for obtaining a service (functionality) from other program modules as well as for exposing a service (functionality) of a program module to the outside. It encompasses connections over a network to remote program modules as well as direct connections based on function pointers to locally available program modules. According to the OSI model, a PlugNode implements a Service Access Point (SAP).

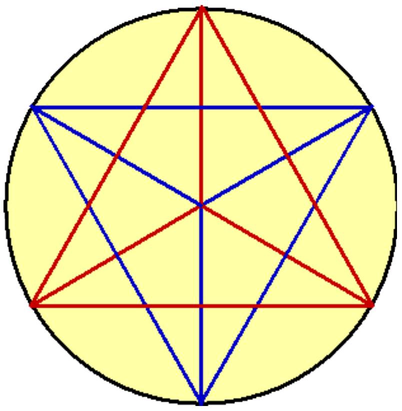
A program module is understood as an instance of a source program which may be written in an arbitrary programming language and which may follow any programming paradigm. Between many others, an instance of a C++ class could become a program module in the sense of this description at runtime if it has been derived from the PlugNode class or if it maintains a member variable of that type.

It may be obvious yet should be admitted here that a different instance from the same source program will become a program module of its own and thus it will communicate to other program modules and in particular with those modules that have been created from the same program source.

Communication structure

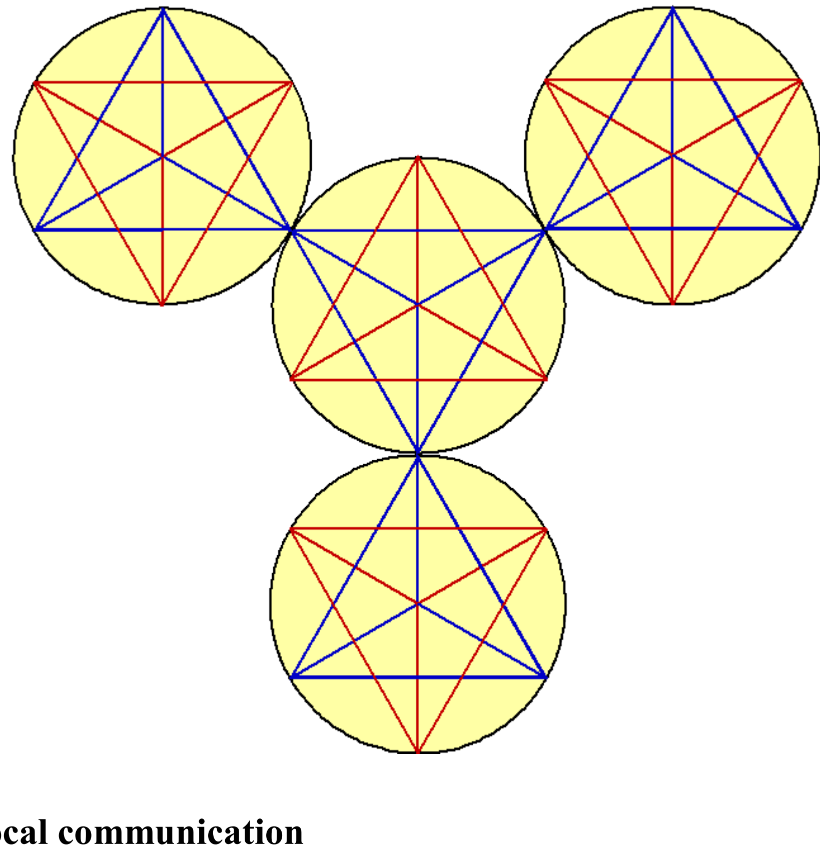
The basic structure for all communication, remote as well as local messaging, should preferably be based on equilateral pyramids (tetrahedron) in order to accomplish optimal information throughput figures. With that design in mind, a current program module is thought to be in the center of two merged tetrahedrons where one tetrahedron is pointing to the front with one of its edges and the second tetrahedron is pointing to the back with one of its edges.

The following pictures have been drawn to facilitate the understanding of a complex communication structure which results from connecting multiple program modules together in a network. By convention, all remote connections will be represented by blue color and the local connections will be represented by red color.



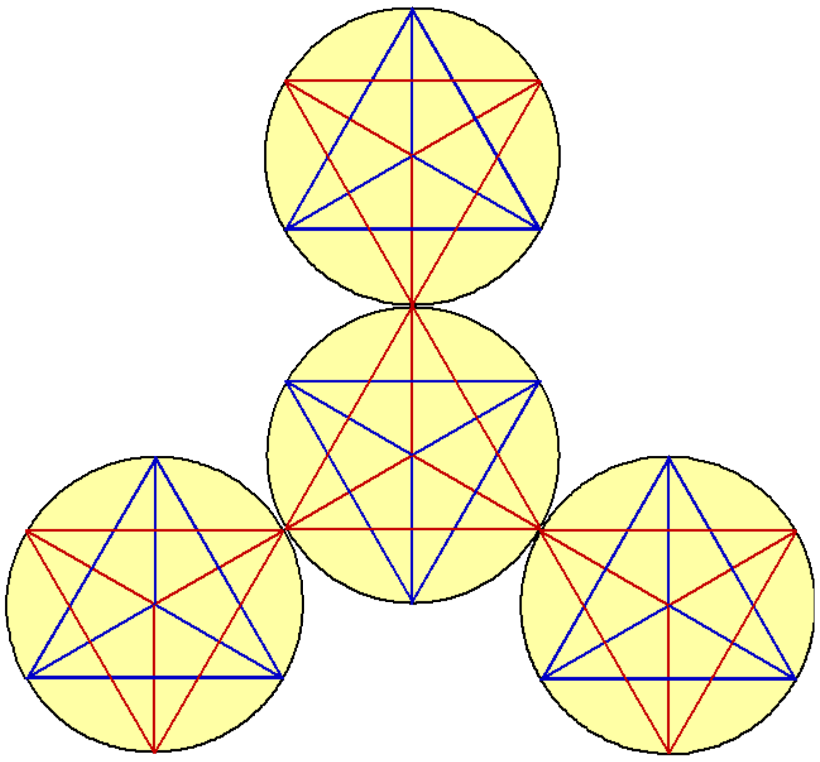
Remote communication

Any edge of a blue pyramid can be connected to a blue edge of another pyramid in order to communicate to that pyramid's program module. The picture shows in a simplified two-dimensional way how a current program module in the center is connected to multiple other program modules on all of its edges. A fourth connected program module is not shown but will be present either in the front or in the back of the center pyramid.



Local communication

In analogy to remote communication, a program module can be connected to upto four locally available other program modules directly. Far more than those one plus four PlugNodes can be connected in a local tetrahedron structure and they will establish a common local messaging group. Once a desired communication peer has been discovered in that local messaging group, a direct and fast connection can be established based on a function pointer. The combination of a flexible messaging structure with the possibility of a direct shortcut provides for an uncompared scalability along with high performance figures.



Generating a network of collaborating PlugNodes

Initial situation

The task is to generate a network of nodes which all have a unique address, e.g. an IP address in the Internet or a memory address in same process space. An address list will not be available! However, all nodes know about the address of a special "first" node, let's call it a "contact node". Further, all nodes may establish connections to upto four other nodes for messaging purposes.

Network formation

Let's assume that the "contact node" is available in the beginning and that it is listening for messages from other nodes that want to connect. A second node knows the address of the "contact node" and sends a CHECKIN request. The first node responds with a CONNECT message and the second node sends a CONFIRM message after which both nodes are connected.

A third node would connect in the same way to the first node. However, after successfully establishing a connection, the third node will be informed about the second node since the first node has local knowledge about that second node and forwards that known address to the third node. Subsequently, the third node attempts to establish a connection also to second node. This way, a small network of three nodes has formed where each node is connected to each other node.

Described procedure may continue until the network contains five nodes since every node can be connected to at most four other nodes. A subsequent sixth node would hit a satured first node when it attempts to CHECKIN to first node. This is not a problem! The first node will DISCONNECT its fourth connection and let the sixth node in. Before, it informs the node on its fourth connection about the address of the sixth node. This way, the new sixth node will be equipped with already two connections. Further, the sixth node will be informed by first node about other known nodes. So the sixth node may try to CHECKIN also to other nodes which obviously will cut some of the already existing connections on those previously connected nodes in the favour of a more inter-connected network structure.

Set of command messages

A small set of messages is used for establishing, confirming, maintaining and shutting down connections to a PlugNode's peers:

START	- a new PlugNode is asked to checkin to a network of PlugNodes
CHECKIN	- a PlugNode asks another PlugNode for establishing a connection
CONNECT	- a new connection has been registered and is pending for CONFIRM
CONFIRM	- a registered connection is confirmed and thereby activated
DISCONNECT	- a PlugNode has shut down a connection
SWITCH	- a PlugNode asks a peer for switching over to a new PlugNode
CONTACT	- a PlugNode informs a new PlugNode about its current peers

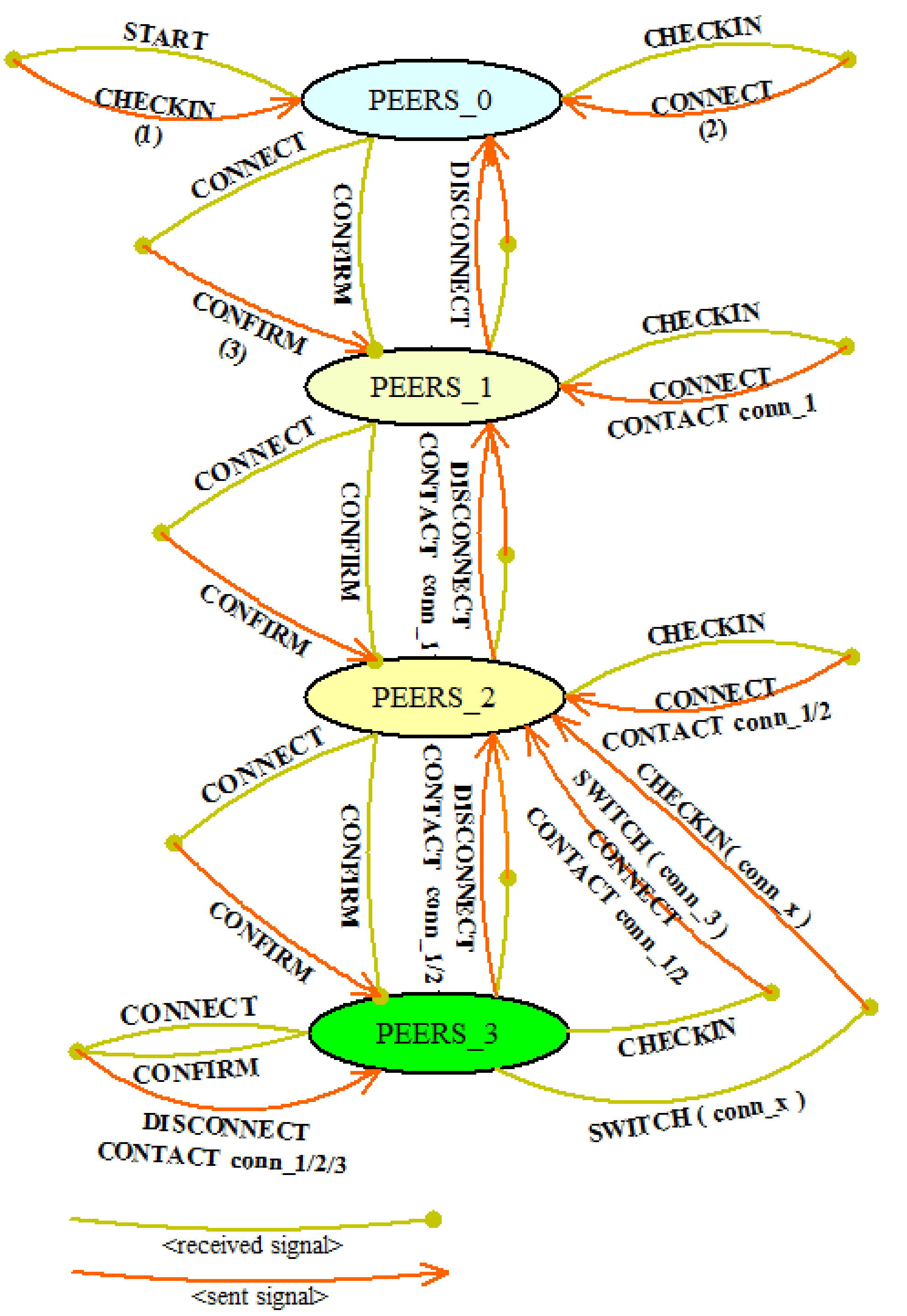
Connection states

A PlugNode's connections will each maintain their unique state dependent on messages that they receive from or send to other PlugNodes. At any time, each connection will have one of the following states:

- STATE_IDLE
- STATE_CHECKIN
- STATE_CONNECT
- STATE_CONFIRM
- STATE_DISCONNECT

Maintaining peer connections by exchanging command messages

The following transitions will be realized in an implementation with upto three peers that aims at maintaining a live PlugNode network structure:



PlugNode_Peers: peer connections changes caused by receiving or sending signals

Usually, a PlugNode's connection starts its lifetime with STATE_IDLE. When a CHECKIN message (1) is sent to another PlugNode then the state changes to STATE_CHECKIN. After a while, the peer PlugNode should respond with a CONNECT message (2). Then the connection that received that message will respond with a CONFIRM message (3) and will go to STATE_CONFIRM. Before the peer connection received the CONFIRM message it will have a STATE_CONNECT and after receiving that CONFIRM message it will also go to STATE_CONFIRM.

Once both PlugNodes' connections reached STATE_CONFIRM, a communication channel is open and will be used for any further communication between that couple of peer PlugNodes.

When a PlugNode wishes to leave a network, then it first releases all connections by sending a DISCONNECT message to each of its peers. The peer PlugNodes will deregister the leaving PlugNode and will change affected connections back to STATE_IDLE after sending as well a DISCONNECT message. The first PlugNode which initiated the disconnection will temporarily be in STATE_DISCONNECT until it received a DISCONNECT message. Then it will also go to STATE_IDLE. Once all connections of a PlugNode reached STATE_IDLE, that PlugNode has successfully left the network and can safely be removed.

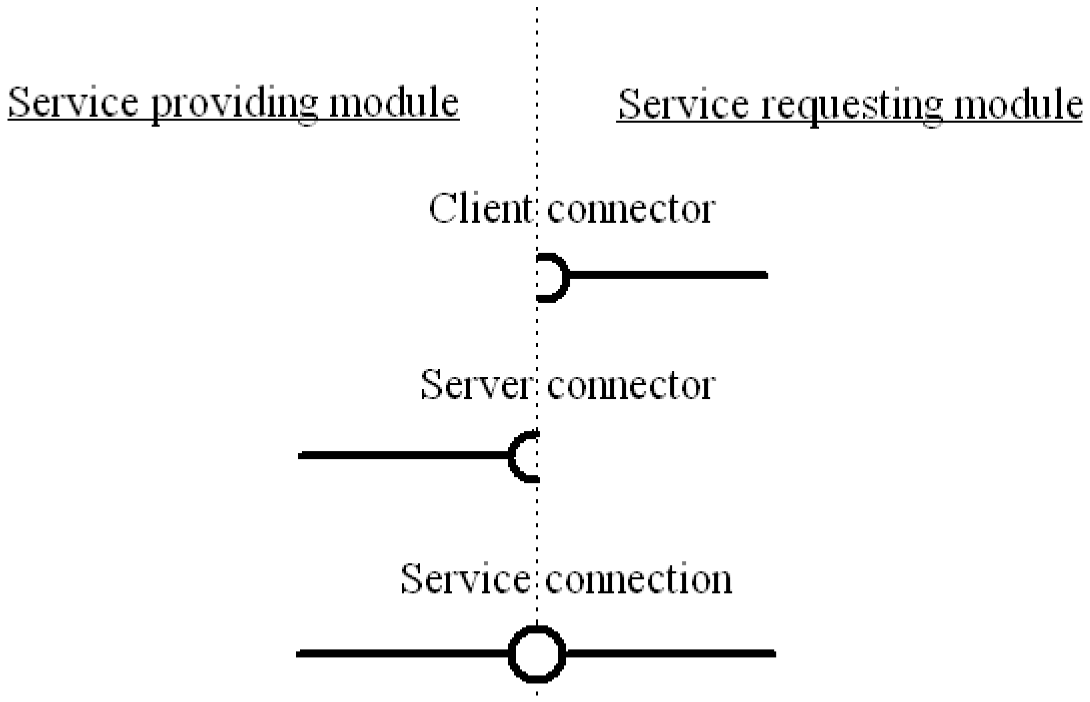
A change of PlugNodes' current network structure is needed whenever a new PlugNode wishes to join the network by contacting a saturated PlugNode that maintains already the maximum of number of valid connections. In that case, the most recently established connection of that PlugNode will be cut in favour of establishing two new connections for previous peer PlugNodes (which are now disconnected from each other). The new PlugNode that wishes to enter the PlugNodes' network structure will be connected to both PlugNodes that just cut their connection in order to allow the new PlugNode to get in to the structure.

A saturated PlugNode will send a SWITCH command to its most recently connected peer in order to accomplish described network change. Both connections of affected peer PlugNodes will go to STATE_IDLE in the result of a sent/received SWITCH message. Afterwards, the first PlugNode (that sent the SWITCH message) will send a CONNECT message to the new PlugNode. The second (previous peer PlugNode) will start initiating a connection to the arriving new PlugNode by sending a CHECKIN message to it.

Whenever sending a CONNECT or DISCONNECT message, a PlugNode will also send CONTACT messages if appropriate. Those CONTACT messages inform the peer PlugNode about currently available peers. This will allow the peer PlugNode to establish more connections in order to get to an as complete as possible set of ideally the maximal number of connections to other PlugNodes participating in that network.

Service connection

Whenever one module uses a service from a second module, those modules are said to be connected by a service connection. In opposite to well known uni-directional service providers like a HTTP server or a local function call, a service connection is a very symmetric setup. The client side will know how to reach the server and the server will know as well about how to reach each client that has subscribed to its service. This symmetry is especially important for a graceful shutdown of a service where all clients have to be informed that the service is no longer available and must not be called any longer. The server will use the client connectors in this case in order to revoke its connector from the clients' list of connectors.



Service handling

When looking at usual implementations for remote and local services, one will most often find solutions that are based on the assumption that a service provider is available almost all the time. A HTTP server in the Internet is an example that is based on that assumption as well as a common function call in a program is based on the same assumption.

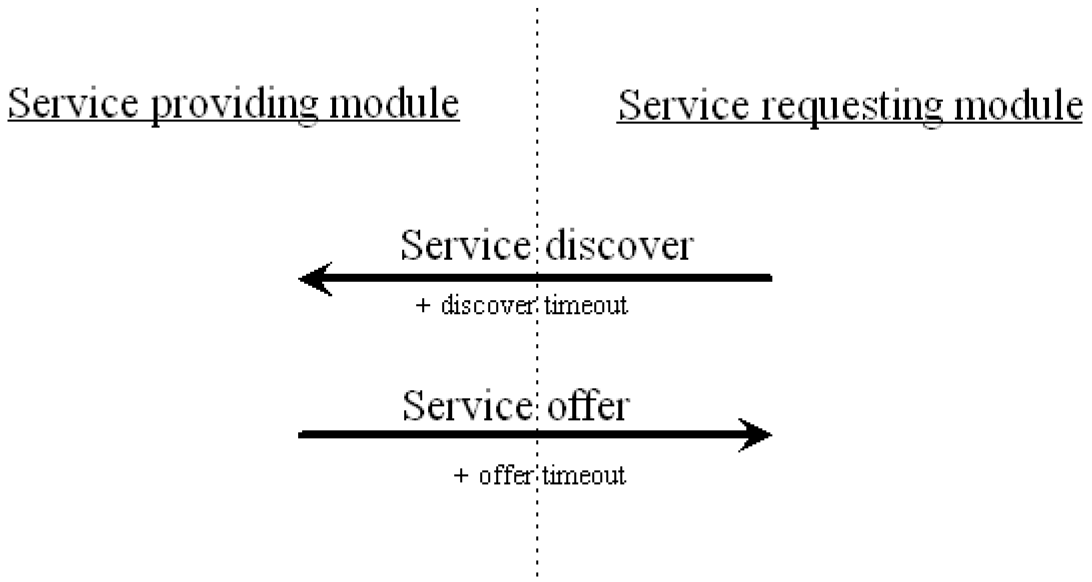
While designing systems based on independent modules, one will become aware that the previous assumption will no longer be valid. Remote modules may shutdown their service while a client is still running as well as a locally available module could stop running while client modules that are connected to it will still continue working. Those roughly described use cases provide to the requirement of a symmetric implementation where each of the two communication partners is allowed to inform the peer about finishing work. The following procedures support such a symmetric approach.

Service arbitration phase

When a client needs a service it will ask the local respectively the remote network of co-workers about a suitable server that is able to provide the wanted service. The client will specify an "inbox" where offers can be sent to. In case of a local service discovery, the "inbox" would be a function pointer which can be called with the offered service's service function as an argument. On the other hand, in case of a remote service the "inbox" would represent an IP address and a port where an information about the offered service's accessibility can be sent to.

Once the client has received one or multiple offers it is said that a temporary service connection has been established. This service connection will be called temporary since either of the service discover and service offer messages will timeout. The timeout value is determined by the sender of the particular message. As long as the service offer timeout has not been reached the service requesting module could decide to accept an offer and would just use the offered service.

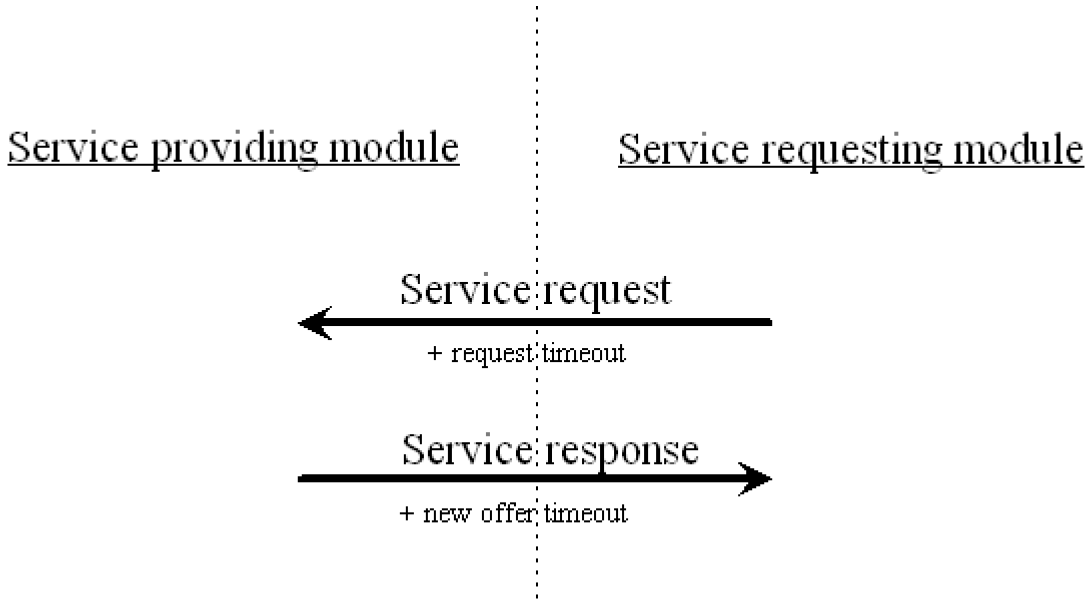
A service providing module will be bound to its offer till the end of the self-specified service offer timeout. On the other hand, it can revoke an offer if circumstances demand for it but only as long as the original service discover timeout has not elapsed since only during that period the service requesting module can be safely reached. Therefore, in a usual setup a service providing module would set the service offer timeout before or equal to the service discover timeout since otherwise it would loose the ability to revoke its offer.



General service request

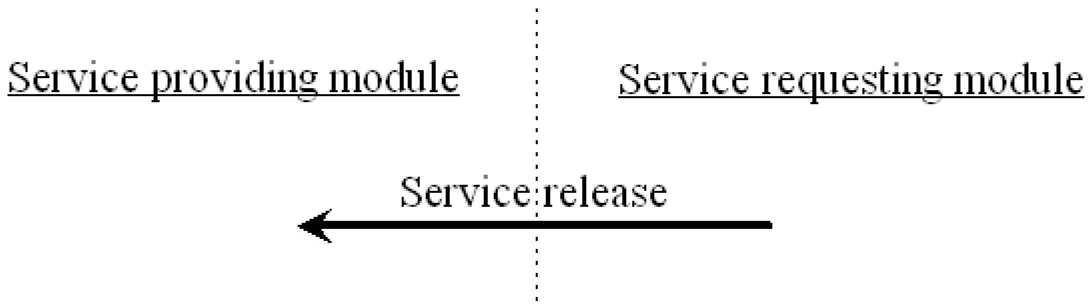
While a valid service connection is present, the initiator of that service connection can use the offered service at any time. For doing so, the service requesting module will send out a service request. The service providing module will generate a response and will send it back to the service requesting module but only if the request timeout has not elapsed yet. Otherwise the response will not be send but will be silently discarded.

When sending a response back to the service requesting module the service providing module can decide to extend the offer for a repeated service. In this case a new offer timeout will be sent together with the response. As already described in service arbitration, an offer can be revoked as long as the service requesting module's timeout (request timeout in this case) has not elapsed.

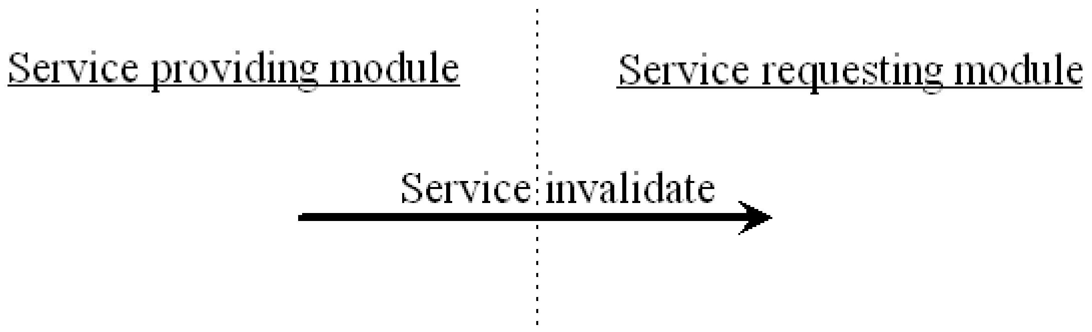


Finishing a service connection

A service connection will stay valid as long as its timeout has not been reached. However, it can be shutdown during its active time from both sides. If a client does not need a service connection any longer it will send a "service release" message to the service provider.



On the other hand, if a server wants to shutdown its service it will send a "service invalidate" message to all peers that still maintain a valid service connection with the service provider at that time.


Service providing module
Service requesting module
Service invalidate

Example for service offers

The range of offered services encompasses everything where one program module may use a service from a second program module. In the following, some examples for those service offers will be given without claiming the listing to be comprehensive.

Implementation of known protocols

- HTTP: Accessing a website / exposing information in a PlugNode network as a website
- FTP: Accessing files by FTP / exposing information in a PlugNode network as a FTP server
- DB: Managing database structures and running database queries
- TCP: Sending/receiving data over a TCP connection
- UDP: Sending/receiving data over UDP
- IP: Sending/receiving IP packets
- Ethernet: Sending/receiving ETH frames

Accessing computer hardware from arbitrary PCs in a network

- USB: Reading/writing information from/to USB devices
- 1394: Reading/writing information from/to FireWire devices
- Parallel port: Reading/writing information from/to parallel ports
- Serial ports: Reading/writing information from/to serial ports
- HD/FD/CD/DVD: Reading/writing information from/to harddisk, disketts, CDs and DVDs
- GUI: Information/image/video stream representation on GUI
- Sound: Audio stream input/output from/to audio hardware

Arbitrary distribution of program functionality locally and to other PCs

- Data conversion into a different format
- Converting data into a suitable GUI representation
- Converting data into audio streams
- Performing calculations

Offering information

- Advertising
- Information about companies/organisations/events/persons
- Information from manually maintained data collections
- Automatically registered measurement data

Open issues

- Time synchronisation for timeout handling
- Information structure for service discovery/requests/responses
- Handling tetrahedron structure when connecting/disconnecting modules
- Efficient message throughput in large module structures
- Managing offered services' access rights

Created: 2006-07-25 by Eckhard Kantz