CS-Prolog II

Version 2.3

Networking

Supplement to User's Manual

ML Consulting and Computing Ltd.

Budapest, Hungary

April 1998

Contents

1. Introduction

This document describes the networking interface of CS-Prolog II. It constitutes a supplement to CS-Prolog II User's Manual.

In order to understand this description, working knowledge of CS-Prolog II real time programming features and built-in predicate set is necessary.

The addition of networking facilities to CS-Prolog II had been accomplished in the framework of the EU funded INCO-Copernicus project *ExperNet: A Distributed Expert System for the Management of a National Network*, No 960114.

ExperNet relies on data provided by the public domain *NAS Hierarchical Network Management System* (HNMS) developed at the NASA Ames Research Center, and also includes source code borrowed from there for interfacing. That's why HNMS has a distinguished status in CS-Prolog II Networking.

The communication model implemented in CS-Prolog II is a moderate generalization of the networking requirements dictated by the ExperNet project where CS-Prolog programs cooperate with partner applications in a distributed system.

2. Networking concepts

As a natural extension of CS-Prolog channel concept, the external communication conceptually consists of unidirectional message streams.

In order to facilitate the speed-up of external communication asynchronous message passing is introduced as an option. Send operation in this case still remains blocking but the condition for continuing execution is the availability of sufficient buffer space instead of the commencement of the matching receive operation.

Communication with applications using different protocols is possible (heterogeneity). At present only the TCP/IP protocol is fully supported as the primary network access facility. Restricted support of the UDP/IP and the HNMP protocols is implemented in the form of *foreign partner* communication. The details of different communication kinds are mostly hidden behind a uniform interface.

An accessible partner is a fellow application we can communicate with. A partner can be an other CS-Prolog application. In this case the communication with it is unrestricted in the sense that any Prolog term can be sent and received. Communication with a non CS-Prolog (foreign) application is more restricted. Each type of foreign applications requires a specific translation between its message format and Prolog terms. This translation is performed by an entity called mediator. The mediator also might be required to perform some protocol-conversion tasks.

We provide the possibility of taking part in the work of an existing distributed system that maintains its network configuration probably by a non CS-Prolog master, using perhaps its own protocol.

For the Prolog programmer the communication environment appears as a homogenous address space (**community**). All partners will be accessed via channel messages. A separate mechanism is introduced for connecting channels to external partners. The most important entity for this task is the so-called **port**.

Every application taking part in a dynamically changing distributed system needs a picture of the current configuration of that system (**subnetwork** for short) to find out which partners are accessible and at what ports. However, in a large subnetwork an application is normally concerned only with a subset of the possible partners, so a narrowed 'sub-map' will be sufficient. CS-Prolog II provides the necessary devices for maintaining such a view.

Besides, or instead of, taking part in a centrally managed subnetwork, an application can have a set of private network connections managed by itself.

This manual describes the second release of networking CS-Prolog II. In this release only the TCP/IP protocol is fully supported, the community can be set up only in non-managed mode and only private network connections can be established to other CS-Prolog partners. A restricted form of external subnetwork management — partner directory service — is available from the HNMS server accessed as foreign partner. Two kinds of non-CS-Prolog applications can be accessed as foreign partners through appropriate mediators: the HNMS server and a broad class of applications that receive and send messages consisting of plain ASCII text lines. There is no support for the common dock yet.

3. Basic networking notions

In CS-Prolog II networking interface an object-like approach is used. The interface objects (enumerated later) have attributes and methods for retrieving attribute values and setting some of them. These attributes describe the properties of the object.

The interface objects are local to the application, even when they represent a global communication object.

Every object has a special attribute named **status**. Status gives information about the state of the object (initializing, normal, closing, etc.).

Each object instance has a name attribute, which must have a restricted atom as value (unique within its object class). This name atom in some cases is fixed in advance, in other cases it is assigned when the object is created. The latter kind can be explicitly specified in the call creating the object instance, or the program can have the system generate a name. System-generated names are atoms having a specific prefix. Explicitly assigned names are subject to the following restrictions: the atom **nil** (`[]`) cannot be used, the name cannot have the specific prefix **`$`** (currency character) as its first character, and the length of the atom must not exceed an implementation defined limit (presently 31 characters).

The built-in predicates used for setting up communication paths to partners across the network are asynchronous by nature; i.e., they do not suspend the calling process. The progress of the actions is reflected in the state of the corresponding interface objects. The user program is notified of the success, failure or other termination of an action through a special real-time event if such an event is specified at initialization. These optional notifications are called **alerts**. Some other asynchronous state changes are also reported to the program using alerts, e.g., when an other application (called unsolicited partner) initializes a network connection to our application by including our application as a private partner into its local community.

We conclude this chapter by some remarks on terminology. For the sake of simplicity we use some well-known terms specifically to denote interface object types, e.g., **port** or **connection**. We hope that most of the time this will not cause any confusion, because this description in general is not concerned with their customary meaning. There are, however, some possible conflicts to note in advance:

> **partner** is an interface object representing a partner application, but sometimes the latter is also referred to as *partner*.
>
> **connection** is an interface object representing an outgoing message stream (similar to a file opened for writing). It is uni-directional. In some cases we also have to talk about network connection, which is a lower level notion, e.g., a TCP/IP connection between partner applications maintained by the supporting software layer. We will refer to the latter kind by the qualified term *network connection* when the context requires.
>
> **port** normally means the interface object to which a message stream (**connection**) can be directed from a partner application. It is different from the port notion defined for the TCP/IP protocol, which we will always call *ip_port*.

Objects of the networking interface will be called network objects for short.

Network object instances will be called simply network objects or just objects when the context allows.

Port and dock are very similar, so instead of using the phrase ‘port or dock’ every time, we usually refer to port only. When the distinction is important, it will be made specifically.

3.1 Network objects

Objects of the following types can be created: community, mediator, partner, port, dock and connection.

All network objects have a name that can be used whenever the object is referenced. The name of the community is fixed (only one community can exist at present). Other names can be supplied by the user or they are generated by the system. The generated names begin with the dollar sign (`'$'`). The object names specified by the user must not begin with this character. The **nil** atom (`[]`) is also excluded from the domain of valid network object names.

Network objects can be removed (closed, deleted) by the application which created them. These deletions can be done gracefully — when the object removal is negotiated with the concerned partner, or forcibly — when the removal is immediate, and only an attempt to notify the affected partner is undertaken.

There is also a special object type called NetPeer, which cannot be created by the application. NetPeer objects are created and controlled by the system if our application is connected to an external subnetwork manager providing directory service. The information about other CS-Prolog applications — peers — connected to the same external manager is stored in these objects. In most respects NetPeer objects behave like the others.

3.1.1 Community

Community is an abstract entity describing the world we communicate with. The community maintains the current configuration of the communication **partners**. The partners can be subdivided into two groups, those from the centralized subnetwork and private clientele. The centralized subnetwork has a managing partner, which takes care of the connections inside it, and maintains the full map of the subnetwork. Joining the centralized subnetwork means connecting the managing partner (**manager** for short).

All partners belonging to the subnetwork can access the full picture of the subnetwork via the manager. However, for efficiency’s sake, each partner manages a restricted local map describing only that part of the subnetwork which is relevant for its task. Changes in the full picture are projected onto each local map. Private partners are included in the local map, but are not noted in the centrally managed full picture.

The community has to be explicitly initialized from CS-Prolog. The initialization of the community depends on the status of our program in the network. This status can be manager or subordinate (slave). Initialization conceptually consists of two stages: creating the community object and populating it with partners (**community_init/5** and **community_activate/1**). Further configuration changes can be accomplished by a separate method (**community_change_config/2**).

Community termination method is implicit on Prolog shutdown, but as part of good programming practice it should be invoked explicitly from the program (**community_shut/1**). On community shut all subordinate network objects (partners, mediators, ports, docks, and connections) are closed.

3.1.2 Partner

A partner is a fellow application we can communicate with. Our own application also can be regarded as a partner for communication purposes. Partners can be created at community initialization (**community_activate/1**) and created or removed any time later while the community exists (**community_change_config/2**). A special partner, denoting our application itself, is always created, it is the **self partner**. The self partner can be used in the same way as external ones.

An important characteristic of a partner is whether or not it understands the message format and exchange protocol used internally by CS-Prolog II. If it does, then we can communicate with it directly. Such partners are termed ***prolog*** partners for brevity. In the opposite case the partner is a ***foreign*** one, and we need the services of a special agent (called **mediator**) during the communication with it.

Outgoing messages can be directed to a selected **port** of a CS-Prolog partner application or to a **dock** offered by a foreign partner. The representation of an outgoing message stream is the **connection**.

The stream of all incoming messages is separated into substreams. Each incoming substream is available at one of the ports or docks created by the program for this purpose. In general several connections are merged into one incoming substream; the contributing connection can be identified by a (newly introduced) argument of the **receive/4** predicate. Ports and docks are essentially the means of demultiplexing the incoming message stream into substreams.

Network connection can also be established implicitly with uninvited partners that contacted us unilaterally; they are the so called **unsolicited partners**. Such partner applications are not represented by partner objects in the community. The total number of partners (either explicit or unsolicited) a CS-Prolog application using TCP/IP protocol can be connected with is limited (currently 30).

A partner can be specified in different ways, but fundamentally it is determined by its net address the format of which depends on the communication protocol used. At present only the IP protocol family is supported where a partner's net address consists of two parts: host address and TCP or UDP port number. The first describes the computer where the partner application can be found and the second part describes the operating system level port number where this application **listens** to the outer world.

3.1.3 Mediator

Mediators are the special agents that enable a CS-Prolog II program to communicate with foreign partners. For each supported kind of foreign partners there must be a specific mediator which translates the message formats and converts the exchange protocols between those used by CS-Prolog II applications on one side, and the foreign partner on the other side.

Each foreign partner requires a separate instance of the appropriate mediator kind as a mediator object.

The detailed specification of the available mediator kinds is contained in chapter 11.

3.1.4 Port

Ports represent incoming message substreams. They are explicitly created and play the role of a sender for a CS-Prolog channel specified at port creation. The other end of the channel can be used in the same way as the receiving end of any internal channel.

Ports are visible for partners. A connection (outgoing message stream) can be directed to a (usually remote) port and a channel is associated with the connection.

There are two kinds of ports: advertised and unadvertised. Names of advertised ports are available from a specific partner attribute while names of unadvertised ports should be acquired directly from messages or agreed upon in advance.

Summarizing the above: a port is connected to a local channel into which it feeds messages coming from possibly several incoming remote connections

At port creation a buffering parameter can be specified indicating the size of message buffer. If it is 0, the remote connection connected to this port will be able to send a new message only after the port accepted the previous one.

3.1.5 Dock

Docks represent incoming message substreams, like ports; they serve foreign communication. One common dock is created when the community is initialized; it serves all non-specific incoming foreign streams. Additional docks can be created for individual foreign partners if desired.

A channel is associated with each dock and is fed by it. The channel associated with the common dock is specified on community creation. The common dock cannot be destroyed. Other docks are associated with a channel at their creation and can be destroyed.

Each dock receives the messages to be passed to the channel associated with it from a **mediator**. The mediator translates the incoming (foreign) message to a Prolog term according to its specification. Mediators are attached to docks at creation time. The common dock’s mediator is a default one.

3.1.6 Connection

A connection object is the representation of an outgoing message stream. Its attributes include the local channel, the partner’s name and the partner’s port (or dock, if the partner is foreign) to where the stream is directed.

The size of the connection’s message buffer can be set at creation. If it is set to 0 (the default value), the connection will not accept the next message from its local channel until the previous one is received at the target port. In this case the corresponding **send** predicate will be blocked in the same way as for local message exchange. If the value of the buffering attribute is greater than zero then more than one message can be stored in the connection buffer, allowing for several send operations to complete without blocking. In this case the size can be changed later, but only to positive values (the non-blocking transmission type must prevail). This option is introduced with the aim of reducing network communication delays. Buffering can be used when the sender is not interested in being exactly synchronized with the receiver.

3.1.7 NetPeer

NetPeers are special objects serving the needs of the subnetwork directory service provided by a distinguished partner application when our application takes part in the work of a team of distributed applications (managed subnetwork). NetPeer objects cannot be created or removed by the application; they are created and controlled by the system when our application is connected to such a subnetwork manager. Information about other CS-Prolog applications — peers — connected to the same manager is stored in these objects.

NetPeer instances may exist only when the subnetwork manager is functioning normally.

The application program can obtain the relevant data about each recognized peer using the **ask_manager/3** predicate.

3.2 Alerts

Alerts are the means of informing our application of some asynchronous events (state changes) that arise in the networking system. Alerts are generated only if the application had specified a real-time event in community initialization for this purpose. If for example the program has a connection to a remote port which is closed by the partner, an alert is (optionally) generated.

The data argument of the generated event roughly describes the cause of the alert and a specifically targeted attribute query can be used to obtain the details. This data argument is a list of the following form:

```
[WhatHappened, ObjectType, ObjectInstance | Other]
```

WhatHappened is the reason for the alert, **ObjectType** is an atom selecting the type of interface object concerned and **ObjectInstance** is the name of the object instance. The **Other** term is a — possibly empty — list providing additional information.

3.3 Network picture

In a centralized subnetwork of CS-Prolog applications managed by a (possibly foreign) manager program the following types of partners can appear for a specific CS-Prolog program:

Private partners; their addresses have to be available in advance in some way for the program (hardwired in the program, obtained from a file, e.t.c.).

Net partners, which have signed up at the manager, and our program included them in its local picture of the network. The address of a net partner is obtained from the manager.

Latent partners (peers), who are known by manager, but our program didn’t include them in its local network picture. The information about latent partners (network address, advertised ports, and some other) is collected in NetPeer objects and can be asked from the manager.

In the present (TCP/IP-based) implementation of the CS-Prolog low-level communication protocol, in order to be able to communicate with a peer, configuration process has to be performed as for private partners. In other words the program has to add explicitly this partner using the **add_private** action of **community_change_config/2** predicate. Local picture is not supported.

So presently the system handles only private partners; subnetwork management is restricted to directory service, which helps in locating peers. From now on when describing the features of partner objects we will not use the notion of ‘local picture of centralized subnetwork’, and refer to net partners as private ones.

In future CS-Prolog versions, if the underlying network layer provides the possibility of communicating with partners with known addresses without building a specific transmission path to them, the necessity of explicitly configuring peers as net partners may be relieved.

4. Network programming

In a CS-Prolog program that communicates with fellow applications through the network, the following main network program steps can be identified:

Creating those processes (only once, in the prelude phase) which will be engaged in networking.

Initializing the community.

Creating the initial set of network objects that will be accessed from outside (ports, mediators with docks).

Populating the community with partners and start actual networking activity.

Making active connections to partners.

Performing (in normal operation mode) the needed tasks, possibly changing the community by adding/removing ports, partners, connections.

Shutting down the community, disconnecting all connections, closing all ports, removing all partners and finally shutting down the community itself.

These steps — except the first — can be repeated in a cycle if desired.

4.1 Initializing the community

The preliminary steps to build a networking community in a CS-Prolog program is to make some processes ready for networking tasks. A real-time process can be charged with the job of handling the incoming alerts. Creation of such a process is not obligatory (but recommended); if no alert event is provided in **community_init/5** the system does not generate the alerts.

In the working phase (after calling **start_processes/0**) the community can be initialized by calling the **community_init/5** predicate. The arguments of this call determine the role of our application in the set of cooperating applications (manager or slave), the description string (which will be seen by partner applications), the alert event, the channel associated with the common dock (where foreign programs can anchor) and a limit that restricts the number of unsolicited partners (to avoid flooding our community with uninvited partners).

The last step of the preparatory phase is the port creation. Ports can be created at any moment before community shutdown, but it is recommended to prepare those ports that our application offers for partners before community activation. So a fellow program will be able to retrieve our advertised port names immediately after our application is configured as partner there, and connections targeted at prepared ports can also succeed immediately.

During this preliminary period the application does not listen to the outside world yet, and so does not react to external requests. When all tasks described above are completed, the community can be activated (see 4.3).

4.2 Creating ports

Ports are created by calling one of the predicates **port_create/[2,3,4,5]**. Five parameters describe the features of the port to be created, of which only the first two must be explicitly supplied in the call; the other arguments are optional with suitable default values defined. The parameters are as follows:

> The name of the port; it can be specified as an atom or a variable. If a variable is given, the system generates a unique name and instantiates the variable with it. If explicit name is supplied, it must conform the general restrictions for network object names.
>
> The channel which accepts messages from the port. A process can open this channel for receive (either before port creation or after it), and read the messages sent to the port by calling the **receive/[2,3,4]** built-in predicate.
>
> The publicity status of the port, can be **`on`** or **`off`** meaning that the port is advertised or not advertised (the default case). The advertised port names are broadcast to all partners, so they can retrieve these port names (using an appropriate attribute handling predicate). Unadvertised ports can be connected only if the partner knows the name by some other means. The publicity status can be changed later with the proper attribute setting predicate.
>
> The buffering limit for the port. The port accepts at least one message from every incoming connection. If the buffering limit is 0 (the default case), a new message is accepted only after the previous one is consumed on the other end of the port's channel (by a **receive** call). If the buffering limit is positive, the port is willing to accept additional messages from the network as long as there is space in the buffer and the total number of buffered extra messages does not exceed the current limit. This parameter is a hint for the CS-Prolog system, how many messages should the port store before blocking the message stream in remote connections.
>
> End marker indication, can be **`on`** or **`off`** (the default case). When this attribute has the value **`on`**, a special end marker term is inserted into the incoming message stream each time when a remote connection directed to this port is disconnected. Its role is similar to the **`end_of_file`** term indicating the end of an input stream: it denotes the end of a network message substream (from a particular connection). The form of this term is:

```
        end_of_message_stream(MODE)
```

> where **`MODE`** is **`graceful`** or **`force`**, showing the mode of disconnect. The CS-Prolog system provides this marker term in error situations, too, when the connection is broken due to some networking error situation.

4.3 Configuring partners

The community can be populated with partners initially when it is activated (**community_activate/1**). Later on new partners can be added and partners can be removed from the community at any time (**community_change_config/2**). In order to access a new partner its communication protocol and its net address must be known. The protocol and the address of a private partner are acquired by the program independently from the CS-Prolog system. For net partners the manager sends these data together with the notification about the appearance of a new partner, or they can be asked from the manager explicitly.

The address specification depends on the communication protocol.

For each prospective partner a configuration list has to be provided in order to define the necessary data for locating the partner on the network, and some other attributes. The members of this list are *configuration options*, i.e., special Prolog terms each representing one configuration option. Almost every option with a value argument has a default value which is used when the option is not specified. Remember that a partner is fundamentally described by its network address and ip_port number, but these data items can be specified in several different ways.

When the community is populated initially, a special partner object representing our own application is always created. Some of the attributes for this object can be specified explicitly (see the **`self`** option below). This partner will be referred to as *self partner* in the sequel. The self partner cannot be removed.

At any time there can be at most one foreign partner for communicating with a HNMS sever (it will be referred to as *the* HNMS partner).

The following configuration options are available for defining a new partner:

`name(NAME)`

> **`NAME`** — an atom or a variable. If an atom is supplied, then it must conform the restrictions for network object names; it will be the name of the partner. If this option is not specified or if **`NAME`** is variable, the system generates a new, unambiguous atom for this purpose. If **`NAME`** is a variable this generated atom is unified with it.

`mediator(MED_NAME)`

> By specifying this option for a partner that partner is qualified as foreign. **`MED_NAME`** — an atom, the name of an existing free mediator which is to provide the facilities necessary for communicating with this foreign partner (and determines its type). The set of the available mediator kinds is not fixed; they are listed in the appendices in chapter 11. (In Release 2.1 **`hnms`** and **`ascii`** are available.)

`protocol(P)`

> **`P`** — the name of the communication protocol to be used for this partner. Presently the following protocols are known by the system: **`tcp`**, **`udp`**, and **`hnmp`**, indicating the TCP/IP, UDP/IP and HNMP protocols, respectively. The acceptable protocol depends on the partner’s type. Normal (prolog) partners use only the **`tcp`** protocol, which is the default for them. The foreign partner created for communicating with the HNMS server uses the **`hnmp`** protocol, and this is the default for it. Foreign partners created for plain text (ascii) communication can use either the **`tcp`** or the **`udp`** protocol; the default for them is **`tcp`**. Note that the **`service`** option below also can implicitly provide the protocol used in ascii communication.

`hostname(HOST)`

> **`HOST`** — an atom indicating the name of partner’s host. It has to be a valid host name known by the operating system. This option is one of the ways by which the network address can be specified. It doesn't have a default value, except for the case of *the* HNMS partner, for which the local HNMS installation defines the default hostname. (See also the **`ip_addr`** option below.)

`ip_addr(ADDR)`

> **`ADDR`** — an atom containing the TCP/IP address of partner's host in dot notation. If both **`hostname`** and **`ip_addr`** configuration options are specified, they must refer to the same host computer. The default value is the primary address of the local host (used only if the **`hostname`** option is not specified either). This option cannot be specified for *the* HNMS partner (see section 11.2).

`service(NAME)`

> **`NAME`** — an atom, the symbolic port identifier known by the operating system. This option is one of the ways by which the ip_port number can be specified (see also **`ip_port`** option below). There is no default value for **`service`**. For ascii type foreign partners if the **`protocol`** option is specified explicitly then only compatible service is looked for, otherwise the service found provides the protocol type, too. This option cannot be specified for *the* HNMS partner (see section 11.2).

`ip_port(PORT)`

> **`PORT`** — an integer, the operating system port number where the partner listens. If both **`service`** and **`ip_port`** are specified the latter one will be used (no check for consistency of these two options). The default value in the current implementation is **`5130`** (used only if the **`service`** option is not specified either). This option cannot be specified for *the* HNMS partner (see section 11.2).

`self`

> Indicates that the config option list defines parameters for ourselves (our application, the self partner). If **`hostname`** and/or **`ip_addr`** option is also given, the network address specified by them must be one of the valid interface addresses of our host computer. The **`service`** or **`ip_port`** option defines the port where our application will listen to the outside world.

`max_retries(M)`

> **`M`** — an integer, the maximum number of failed net connection attempt retries. The default value is the CS-Prolog maximal integer (practically infinity).

`retry_delay(N)`

> **`N`** — an integer specifying the length of the time-interval (in hundredths of seconds) to wait before the next attempt when a failed net connection is to be retried. The default value is **`6000`** (one minute).

`fcaa_freq(F)`

> **`F`** — an integer indicating the frequency of failed net connection attempt alerts. If a connection request fails, the CS-Prolog system may generate an alert notifying the program about the failure. This option means that the alert will be generated on every **`M`**-th failed connection attempt only. The default value is **`1`** (every failure will be reported).

If the list specifying parameters for a particular partner contains multiple occurrences of an option then only the last one is used, but of course all items are checked for validity. The remaining list is then checked for internal consistency, but no validation against the current community configuration is performed at this stage. Among the internal consistency criteria the following are not self evident: **`self`** can be specified at most once and only in the initialization phase, and **`self`** cannot be foreign.

If all partner configuration lists specified in the current call pass these tests, then the partner objects are created and configuring action is started for each requested partner individually (and asynchronously).

The first step of the configuring action is verifying whether the requested partner has already been configured. If so, the newly created partner object is assigned zombie status and a special attribute is set to the name of the already existing partner object, and optionally an alert is generated.

A special check is performed for a foreign partner supported by a **`hnms`** mediator (*the* HNMS partner). The situation when the configuration of a new HNMS-type foreign partner is attempted while *the* HNMS partner still exists, is considered a conflict exactly like if the same (full) network address had been specified or implied (i.e. it leads to the *partner_already_exists* alert and status change as the case above), even if a different HNMS server is implied.

After this the availability of a communication slot is checked, and if this fails, then the attempt is aborted like in case of fatal network error condition (see below). Otherwise, when all conditions are met, then the setting up of the appropriate net connection begins.

If an attempt to set up a net connection to the desired partner application is not successful the created partner object remains in initial state. Establishing the net connection will be retried no sooner than after **`retry_delay`** hundredths of seconds. If the **`max_retries`** count is reached or a fatal error occurs the operation is aborted and the partner object's status becomes zombie.

The following configuration options are valid for partner removal:

`name(NAME)`

> **`NAME`** — an atom, the name of the partner to be removed.

`graceful`

> Indicates graceful removal, in cooperation with the partner application. All connections directed to this partner will be disconnected in graceful mode before deletion of the partner object, and also all remote connections coming from this partner will be instructed to disconnect.

`force`

> Indicates forced removal. An attempt to notify the partner application will be undertaken, but the local partner object will be deleted anyway. All connections to this partner will be disconnected in force mode.

4.4 Connecting to partners

If an application wants to send a message to another one, a **connection** has to be created (**connect_to_port/[4,5]**, **connect_to_dock[4,5]**). The partner object representing the partner application must already exist. The following five parameters define a connection:

> The name of connection. It can be specified as an atom or a variable; in the latter case the system generates a name for the connection and unifies it with the variable.

> The name of the partner we want to connect to.

The name of the partner’s port which will receive the message stream. This port has to exist in the partner’s community.

Local channel name which will feed messages into the connection. A local process can open this channel for send (either before creating the connection or any time later) and send messages using the **send/2** predicate.

The maximum number of buffered messages. If this optional parameter is 0 (this is the default case), sending a message through this connection blocks the active process, which can resume execution only when the acknowledgment arrives from the receiver process. For positive values of this buffering limit no acknowledgment is awaited and the connection stores the messages as long as there is space in the buffer and the number of unsent messages does not exceed the current limit. Otherwise the next send operation will block, and the block is released when a message can be sent from the connection (the remote port can accept the message).

If the partner is in initial state yet (the net connection is in progress) the requested port or dock connection is suspended, and will be performed when the partner’s status becomes normal. If the partner's status is, or becomes, zombie before the connect request completes, the status of the created **connection** object also becomes zombie and optionally an alert is generated.

4.5 Closing activities

Every network object created by an application can be removed (closed, deleted) by it. These deletions can be done in either of two modes: **`graceful`** or **`force`**. Graceful close means an attempt to clean up all remote links of the object in cooperation with the interested partners. The delivery of all messages that are already on their way is awaited. (In case these messages are not delivered — e.g., because of a network error — the graceful closing will never complete.)

A forced close means the deletion of the object and its links independently from the partner applications; this will always succeed within a moderate amount of time. An attempt to inform the affected partners is undertaken, but no acknowledgment is awaited from them. Pending messages (if any) are silently discarded.

Network objects that cease to function normally because the link to their corresponding remote entity is lost, are not deleted; they become **`zombie`** instead. The application can get rid of these zombie objects removing them with an appropriate predicate. A partner object can become zombie if the remote community disappears either because of a shutdown or a network error. A connection object turns zombie if its remote port is closed or if its partner object becomes zombie.

Graceful port and dock closing induce the following activities:

Every remote connection directed to this port is instructed to cease activity and become zombie after the last buffered message is sent and acknowledged.

Delivery of all pending messages and the disconnection of all affected connections is awaited.

When the last connection is removed the receiving channel is closed and the port or dock is deleted.

An alert is generated (if needed).

Graceful disconnection means the following activities:

The sending channel is closed.

The messages in the connection's buffer are sent (if any).

The remote port or dock is informed about disconnect.

When the acknowledgment arrives from the remote port (following all messages buffered at the port prior to the notification) the connection object is deleted.

An alert is generated (if needed).

Graceful removing of a partner object generates the following actions:

All connections are gracefully disconnected.

When there are no more connections for the partner object, it is deleted.

An alert is generated (if needed).

Graceful community shut down means the following:

All partners (except the self partner) are gracefully removed.

All ports and docks are gracefully closed.

When there are no more ports, docks, and remote partners, the community is deleted.

An alert is generated (if needed).

Closing actions in force mode act in a similar way, with the differences that all implied sub-actions are issued in force mode, there is no waiting for completion, and no alerts are generated (force mode closing always succeeds within a short time).

4.6 Working with foreign partners

Foreign applications do not understand the message format used in prolog-to-prolog communication, so they need an agent that performs the appropriate data and protocol conversion. Such agents are called mediators in our system. They are considered to be part of the CS-Prolog II run-time program, just as the network driver is.

At present there are two mediators defined. The one for plain text communication is denoted by the mnemonic keyword **ascii**. The other, for communicating with the HNMS server, is named **hnms**.

Conceptually a local mediator is communicating with a remote mediator, hosted at the foreign partner, addressing the dock it offers. Data sent by the remote mediator is accepted at the dock of the local mediator. In fact, however, the functionality of the remote mediator is also implemented locally and the remote dock is a fiction that abstracts the capability of the foreign application that it can be connected to.

Docks are similar to ports. They play the same role in the communication; the difference is in the way a dock is prepared for work and is connected to implicitly by the mediator on behalf of the foreign partner. Docks (and mediators) are non-reusable objects in the sense that once they had been dedicated to a specific foreign partner they cannot be used with another, even if the original partner is removed and the new candidate is of the same type. In such circumstances the program has to close the old objects and create new ones.

At present, foreign partners envisioned accept a single, uniform message stream only. This means in our terms that they offer us only one default dock, which need not be named specifically in the **connect_to_dock** predicate. Nor are local dock names advertised like port names.

For generality, however, the foreign dock name argument is required in **connect_to_dock**, but it must specify the default dock of the foreign partner by giving the **`nil`** atom as argument value.

In order to configure a foreign partner, the application program shall first create a dock (unless it wants a silent partner), then create a mediator of the appropriate kind, naming a free dock for it, and finally configure the desired foreign partner, naming the mediator in the list of the configuration parameters (implying thereby that the partner is foreign).

Once the foreign partner is successfully created, the procedure to follow in message exchange is almost the same as for any prolog partner.

The program can issue **connect_to_dock** calls to prepare channels for sending messages to the foreign partner (dock_name should be given as **`nil`**), and can receive messages coming from the foreign partner on the channel attached to the dock of the mediator of the partner.

If there are more than one active connections to the same foreign partner, the messages are merged into one outgoing message stream so that the relative order of messages sent via each individual connection is preserved.

The most important restriction in communicating with foreign partners is in the set of rules specifying that what kinds of prolog terms are accepted and produced by them. This specification constitutes the description of the individual mediator kinds.

The detailed description of the available mediators is contained in chapter 11.

4.7 Status changes of network objects

During its lifetime the community status can have the following values:

initializing	from **community_init/5** until **community_activate/1** completes;
normal	after **community_activate/1** succeeds until **community_shut/1** request;
zombie	after **community_activate/1** fails until **community_shut/1** (self partner is still accessible).
closing	from **community_shut/1** request until the completion of the operation;

A partner object can have the following status values:

initial	from creation until the completion of the net connection process;
normal	from successful completion of the net connection process until the program removes the partner (**community_change_config/2** or **community_shut/1**), or the partner application shuts down, or the connection to it is lost;

zombie	if the connection process failed, or if the partner application had been shut down or if connection is lost to the partner application, until explicit partner removal call.
closing	from explicit partner removal call until the completion of closing;

A port object can have the following status values:

initial	only for advertised ports, from creation until the successful broadcast of port name to partners;
normal	for unadvertised ports is set on creation, for advertised ones when the port name broadcast is done, until port close request;
closing	set when port close request is issued (**port_close/[1,2]** or **community_shut/1**) until completion of the request.

A dock object can have the following status values:

initial	indicates that the dock is free (immediately after creation);
normal	set when the dock is attached to a mediator;
closing	set when dock close request is issued (**dock_close/[1,2]** or **community_shut/1**), until its completion.
zombie	indicates that the partner with which this dock is associated had been removed or became zombie, and there are no buffered messages waiting delivery at the dock.

A connection object can have the following status values:

initial	from connection request (**connect_to_port/[4,5]**) until completion of the request;
normal	from successful completion of the connection process until explicit disconnect request (**disconnect/[1,2]** or removal of the target partner or **community_shut/1**), or until the connected partner closes the target port (or shuts down), or partner is lost;
zombie	if the connection request fails, or if the partner application closes the destination port of the connection, or partner is lost, until explicit closing succeeds.
closing	from explicit disconnect request until its completion;

A mediator object can have the following status values:

initial	indicates that the mediator is unattached (set on creation and also when the hosting partner is removed and the mediator becomes free again);
normal	set when the mediator is attached to a designated partner;
zombie	indicates that the associated dock is zombie.

4.8 Attribute handling

The attributes of network objects describe their properties relevant for the user program. Two sets of built-in predicates serve for retrieving and setting attribute values. Most of the attributes are read-only, i.e., cannot be changed by the user.

The **<*object_name*>_current_attribute/[3,4]** predicates serve for obtaining attribute values. (**<*object_name*>** can be one of: **community**, **partner, mediator**, **port**, **dock**, **connection**.) The first three arguments are:

object name

attribute name

attribute value

They all can be instantiated or uninstantiated. The predicates are resatisfiable, they will enumerate all object name, attribute name, attribute value triplets that match the arguments. The optional fourth argument can be **`conditional`** or **`unconditional`**. It influences the system's behavior if either the community does not exist or the object name is provided in the first argument, but there is no object with this name. Unconditional operation (this is the default) will generate a CS-Prolog exception in these circumstances, while the conditional version will simply fail.

The modifiable attributes can be set with the predicate **<*object_name*>_set_attribute/3**. The arguments are the same as arguments of **<*object_name*>_current_attribute/3**, but in this case they all must be instantiated.

5. Object attributes

In this section the attributes of network objects are listed. Attributes describe the relevant properties of an object. Some attributes are fixed at creation (and retain their value during the lifetime of the object), others can change. There are read-only (modified only by the system, if at all) and user-modifiable attributes.

5.1 Community attributes

Name	Value	Variance
name	net object name atom	fixed
mode	**manager** or **slave**	fixed
description	description string	fixed
manager	one of the partners or **nil**	fixed
common_channel	channel name for common dock or **nil**	fixed
alert_event	event for alerts or **no_event**	fixed
host_name	local host name	fixed
status	**initializing**, **normal**, **closing**, **zombie**	changing

5.2 Partner attributes

Name	Value	Variance
name	net object name atom	fixed
protocol	protocol name (**tcp**, **udp**, or **hnmp**)	fixed
host_name	initially **nil**, set to the host name taken from partner's community when status becomes normal	changing
ip_addr	net address (string in dot notation) or ‘0.0.0.0’ (for the HNMS partner)	fixed
ip_port	low level port number (65535 for the HNMS partner)	fixed
description	string (atom) from the community of the represented application	fixed
type	communication level (**prolog** or **foreign**)	fixed
port_names	list of partner’s advertised port names	changing
net_status	**private** or **net**	fixed
retry_delay	time interval before next retry if net connection fails	modifiable
fcaa_freq	frequency of alerts if net connection fails	modifiable
max_retries	number of retries if net connection fails	modifiable
self	**true** for the self partner, otherwise **false**	fixed
other_partner	initially set to **nil**, if conflict with already configured partner is detected then changed to the name of that partner	changing
status	**initial**, **normal**, **closing**, **zombie**	changing
mediator	name of the associated mediator	fixed

5.3 Mediator attributes

Name	Value	Variance
name	net object name atom	fixed
kind	**hnms** or **ascii**	fixed
description	description string	fixed
partner	the name of the partner to which the mediator is attached, or **nil**	changing
dock_in	the name of the attached dock or **nil**	fixed
flag1	(depends on kind **-** set at creation from argument)	fixed
flag2	(depends on kind **-** set at creation from argument)	fixed
status	**initial**, **normal, zombie**	changing

5.4 Port attributes

Name	Value	Variance
name	net object name atom	fixed
channel	receiver channel name	fixed
connections	list of incoming remote connection descriptors	changing
queue_length	number of unprocessed buffered messages	changing
buffering_limit	upper bound for the number of messages that can be buffered	modifiable[1]
advertised	publicity status (**on**, **off**)	modifiable
insert_end_marker	**on** or **off**	fixed
status	**initial**, **normal**, **closing**	changing

5.5 Dock attributes

Name	Value	Variance
name	net object name atom	fixed
channel	receiver channel name	fixed
queue_length	number of unprocessed messages	changing
buffering_limit	upper bound for the number of messages that can be buffered	modifiable[2]
mediator	name of the mediator to which the dock is attached, or **nil**	changing
status	**initial**, **normal**, **closing, zombie**	changing

[1] If the buffering argument on port creation is positive, it can be changed later to an other positive number, but cannot be changed to zero. Zero initial values cannot be changed at all.

[2] If the buffering argument on dock creation is positive, it can be changed later to an other positive number, but cannot be changed to zero. Zero initial values cannot be changed at all.

5.6 Connection attributes

Name	Value	Variance
name	net object name atom	fixed
partner	name of the connected partner	fixed
target name	name of the connected port or dock (at partner)	fixed
channel	sender channel name	fixed
queue_length	number of unsent buffered messages	changing
buffering_limit	upper bound for the number of messages that can be buffered	modifiable[3]
status	**initial**, **normal**, **closing, zombie**	changing

5.7 NetPeer attributes

Name	Value	Variance
name	net object name atom	fixed
description	string, from the peer application's community	fixed
protocol	protocol name (now only **tcp**)	fixed
host_name	taken from peer application's community	fixed
ip_addr	net address (string in dot notation)	fixed
ip_port	low level port number	fixed
port_names	list of port names advertised by the peer	changing
partner	the name of the partner object if the peer is configured as a partner, otherwise **nil**	changing

[3] If the buffering argument on connection creation is positive, it can be changed later to an other positive number, but cannot be changed to zero. Zero initial values cannot be changed at all.

6. Alerts

This chapter contains the detailed description of networking alert events. Alerts are optionally sent by the system to inform the CS-Prolog program about asynchronous events that occurred in connection with some networking operation. The event name used for alerts can be set on community initialization. The event data part of an alert is a list describing the alert itself. This list has at least three elements:

Alert name — an atom indicating the reason of the alert;

Object type — the name of the object type related to the alert;

Object instance — the name of the specific networking object (of type **Object type**) related to the alert.

There may be further elements providing additional information specific to the alert in question. When *error number* and *error text* are included in this part, they represent the corresponding data items obtained from the operating system.

6.1 Community related alerts

Alert name	Object type	Object instance indicated	Other	Explanation
community_activate_failed	community	the community	error number, error text	An error occurred while activating. E.g., wrong tcp port number was given (self partner is still accessible).
community_shut_succeeded	community	the community	**nil**	The shutdown has been completed

6.2 Partner related alerts

Alert name	Object type	Object instance indicated	Other	Explanation
new_partner	partner	the new partner object	**nil**	The net connection to the partner has been successfully set up.
hnms_partner_is_recovering	partner	the hnms partner object	**nil**	The hnms driver lost connection to the hnms server and is trying to reconnect.
hnms_partner_recovered	partner	*the* hnms partner object	**nil**	The hnms driver succeeded in re-establishing connection to the hnms server.
partner_already_exists	partner	the failed partner object (zombie)	conflicting partner's name	The requested partner is already configured.
net_connection_failed	partner	the failed partner object (zombie)	qualification, error number, error text	Some error occurred. Qualification can be: **still_in_progress**, **retry_count_reached**, **fatal_error, hnms_driver_launch_error, hnms_driver_does_not_respond,**
partner_removed	community	the community object	ip_addr and ip_port config options	Graceful partner removal succeeded. The other elements identify the removed partner.
Unsolicited_partner	community	community object	ip_addr and ip_port config options	An other application has set up a net connection to us. The **other** elements identify this application.
partner_is_closing	partner	partner object (zombie)	**nil**	The partner application shuts down its community.
partner_is_lost	partner	partner object (zombie)	error number, error text	The net connection to the partner is broken due to an error.
partner_limit_exceeded	partner	partner object (zombie)	**nil**	There is no available free communication slot.

6.3 Port and Dock related alerts

Alert name	Object type	Object instance indicated	Other	Explanation
port_close_succeeded	community	the community object	port name	The graceful port closing has been completed.
dock_close_succeeded	community	the community object	dock name	The graceful dock closing has been completed.

6.4 Connection related alerts

Alert name	Object type	Object instance indicated	Other	Explanation
connect_to_port_succeeded	connection	connection object	**nil**	Connection to a remote port has been successfully completed
connect_to_dock_succeeded	connection	connection object	**nil**	Connection to a remote dock (of a foreign partner) has been successfully completed
connect_to_port_refused	connection	connection object (zombie)	**nil**	Connection to a remote port has failed. (No such port, partner is closing, etc.)
connect_to_dock_refused	connection	connection object (zombie)	**nil**	Connection to a remote dock has failed. (No such port, partner is closing, etc.)
disconnect_succeeded	community	community object	connection name	The graceful disconnect request has been completed.
remote_port_is_closing	connection	connection object (zombie)	close mode (force or graceful)	The connected remote port is being closed by the partner.
remote_dock_is_closing	connection	connection object (zombie)	close mode (force or graceful)	The connected remote dock is being closed by the partner.

6.5 NetPeer related alerts

Alert name	Object type	Object instance indicated	Other	Explanation
appeared	net_peer	net_peer object	**nil**	A new peer application appeared (not represented previously).
reconnected	net_peer	net_peer object	**nil**	A peer application known about previously has reconnected the network manager. The continuity of its operation might have been disrupted.
updated	net_peer	net_peer object	**nil**	The advertised portname list of the peer application has changed.
net_peer_terminated	community	the community object	ip_addr and ip_port config options	The net peer application has signed off from the subnetwork manager (either shut down gracefully, or removed the subnetwork manager from its community).
net_peer_removed	community	the community object	ip_addr and ip_port config options	The subnetwork manager lost connection to this peer without the peer having previously signed off.
net_peer_lost	community	the community object	ip_addr and ip_port config options	The NetPeer object for this peer had been discarded because connection is (temporarily) lost with the subnetwork manager.
implementation_limit_exceeded	community	the community object	**'directory_entry_size'**	The total length of strings contributed to the NetPeer descriptor entry of this community is too large.

7. Exception terms

The following new exception constants are introduced in connection with the networking predicates:

`ValidDomain`
```
        net_close_mode_option
        net_init_mode_option
        net_partner_limit
        port_name
        option
        netconfig_option
        explicit_netobj_name
        ip_addr
        ip_port
        netconfig_action
        net_attribute
        net_attr_value
        predicate_mode
        mediator_kind
        hnms_option
```

`ObjectType`
```
        netobject
        nethost
        netservice
```

`PermissionType`
```
        netobject
        netobj_name
        net_attribute
        net_max_qlength
```

A new exception term and an associated error type category are also introduced:

```
consistency_error(ConflictType, Culprits)
```

Two options given in a built-in predicate for an object contradict. The **`Culprits`** term has the form **`C1 + C2`** indicating the two conflicting options.

`ConflictType`
```
        netaddr
        selfhost
```

8. Networking built-in predicates

The new networking built-in predicates ought to be called in the working phase (after the call of **start_processes/0**). These predicates are non-backtrackable; i.e., their effect is not undone if the Prolog program does a backtrack over them.

community_init/5

Description

```
community_init(Mode, Description, DockChan, AlertEvent,
               Limit)
```

Mode is the role of the community in the centralized subnetwork: **`manager`** or **`slave`**. **Description** is an arbitrary string that can facilitate fellow applications in identifying us. **DockChan** is a channel name accepting the messages from foreign partners directed to the common dock. It can be **nil** meaning that these messages should be silently discarded. The channel can be non-existent in the moment of the call, but if it does exist, the sending end must be free (as for an unconditional **open_channel_for_send** call). The **AlertEvent** argument has to be an existing event name or the special atom **no_event**. The alerts generated for various asynchronous network-related state changes are directed to the real-time process that specified this event. If **no_event** was supplied then alerts are not generated at all. With the **Limit** argument the system is instructed to reserve this number of communication slots for configuring partners. The maximum number of communication slots is limited (at present this limit is 30), and these slots can be used by unsolicited partners. The CS-Prolog system will guarantee that at least **Limit** explicit partners can be configured. Both active partner creation and unsolicited partner connection will be rejected if there is no free communication slot available. Specifying zero for **Limit** means reservation for explicit partners half of the available communication slots (presently 15). Note that after a partner's removal the corresponding communication slot still may remain occupied by the net connection (if the partner application had also configured us on its side), which in this case is rendered *unsolicited*. So Limit gives us guarantee only for the possibility of the first configurations, but does not reserve these slots as usable for explicit partners only.

Template and modes

```
community_init(+net_mode, +atom, +channel, +event,
               +integer)
```

Examples

```
community_init(slave, ’Test program’, [], netalert, 0).
```

The community will be initialized in slave mode. No foreign messages are accepted at the common dock, the networking alerts will be directed to the process reacting to the **`netalert`** event.

Errors

`permission_error(parallel, process, 0)`
The built-in predicate has been called in the prelude phase.

`instantiation_error`
One of the arguments is a variable, or **DockChan** contains a variable.

`system_error`
The community is already initialized. **ErrInfo-Other** will be the atom **community_already_initialized**.

`type_error(atom, Mode)`
The **Mode** argument is neither a variable nor an atom.

`domain_error(net_init_mode_option, Mode)`
The **Mode** argument is not valid.

`type_error(atom, Description)`
The **Description** argument is neither a variable nor an atom.

`domain_error(unique_name, DockChan)`
DockChan is not a valid unique name.

`permission_error(open, channel, DockChan)`
The **DockChan** channel is already opened for send. The **ErrInfo-Other** will be the atom **already_open**

`permission_error(open, channel, DockChan)`
The **DockChan** channel is already opened for receive by a connection. The **ErrInfo-Other** will be the atom **already_network**.

`domain_error(unique_name, AlertEvent)`
The name of the event is not a valid unique name.

`existence_error(event, AlertEvent)`
The required event does not exist.

`permission_error(access, event, AlertEvent)`
AlertEvent is one of the reserved system event names.

`type_error(integer, Limit)`
The **Limit** argument is neither a variable nor an integer.

`domain_error(not_less_than_zero, Limit)`
Limit is an integer that is less than zero

`domain_error(net_partner_limit, Limit)`
Limit exceeds the permitted number of partner communication slots (30).

community_activate/1

Description

```
community_activate(ConfigOptionLL)
```

The predicate activates the community initiating the creation of partners described in **ConfigOptionLL**. After the call of **community_activate/1** the application will be ready to answer partner requests. **ConfigOptionLL** has to be a list whose members are lists of valid new partner config options (see 4.3).

Self partner is always created (with default attributes if not explicitly present).

Template and modes

```
community_init(+config_option_list_list)
```

Examples

```
community_activate([
   [self, ip_port(5555)],
   [name(friend), hostname('eric.ml-cons.hu'), ip_port(5556)]
                  ]).
```

The community is activated. Our application will listen on TCP/IP port **`5555`**. One remote partner is created, it is looked for on the host **`eric.ml-cons.hu`** , expected to be listening on port **`5556`**.

Errors

`system_error`
The community is not initialized. **ErrInfo-Other** will be the atom **community_not_initialized**.

`instantiation_error`
ConfigOptionLL or one member of this list is a variable.

`type_error(List, ConfigOptionLL)`
Either **ConfigOptionLL** or one of its members is not a list.

`domain_error(net_config_option, Opt)`
.

`domain_error(explicit_netobj_name, Name)`
Name, given in a **name(Name)** option for an object to be newly created is not allowed as a user-specified name for a net object. It is **nil** or begins with the reserved character (**'$'**), or its length exceeds the maximum allowed for network object names (31).

`permission_error(create, netobj_name, Name)`
Name, given in a **name(Name)** option is already the name of an existing partner. **ErrInfo-Other** will be the atom **partner**.

`existence_error(nethost, Host)`
Host, given in a **hostname(Host)** option is unknown.

`consistency_error(selfhost, self + Host)`
Host, given in a **hostname(Host)** option for the self partner is not valid for our host.

`consistency_error(selfhost, self + mediator)`
mediator(M) option, which qualifies the partner as foreign, is invalid together with the option **self**, which denotes ourselves (the application program running that executed the operation).

`consistency_error(netaddr, Protocol + Kind)`
Protocol, given in a **protocol(Protocol)** option and **Kind**, the kind of the mediator specified in the **mediator(M)** option, are contradicting. **ErrInfo-Other** will be the atom **protocol_is_invalid_for_this_mediator**.

`consistency_error(netaddr, Protocol + prolog)`
Protocol, given in a **protocol(Protocol)** option is not accepted for normal (prolog) partners. (At present only tcp can be used in communication with prolog partners.)

`domain_error(ip_addr, Addr)`
Addr, given in an **ip_addr(Addr)** option is not a valid TCP/IP address.

`consistency_error(netaddr, Host + Addr)`
Host, given in a **hostname(Host)** option, and **Addr**, given in a **ip_addr(Addr)** option, are contradicting.

`existence_error(netservice, Service)`
Service, given in a **service(Service)** option is unknown.

`domain_error(not_less_than_zero, TcpPort)`
TcpPort given in a **ip_port(TcpPort)** option is an integer less than zero

`domain_error(ip_port, TcpPort)`
TcpPort given in a **ip_port(TcpPort)** option exceeds the limit for valid port numbers.

`permission_error(modify, netobject, self)`
There are more than one **ConfigOptionLL** members for the self partner.

community_change_config/2

Description

```
community_change_config(Action, ConfigOptionLL)
```

The predicate adds partners to, or removes partners from, the community depending on **Action**. **Action** can be **`add_private`** or **`remove`**. **ConfigOptionLL** has to be a list whose members are lists of partner config options, valid for **Action** (see 4.3).

A request to remove the self partner is silently ignored.

Template and modes

```
community_change_config(+config_action,
                        +config_option_list_list)
```

Examples

```
community_change_config(add_private, [
   [name(near), ip_port(9992)]
   [name(X), hostname('enterprise'), service(cspnet)]
                                      ]).
```

Two partners are added to the community. The first will have the name **`near`**, it resides on the local host. The other one is on host **`enterprise`**, its name will be generated by the system, and passed back to the caller in the variable **`X`**.

```
community_change_config(remove, [[name(near)]]).
```

The partner named **`near`** is removed from the community.

Errors

`system_error`
The community is not initialized. **ErrInfo-Other** will be the atom **community_not_initialized**.

`permission_error(create, netobject, Action)`
The predicate was called prior to **community_activate/1** call. **ErrInfo-Other** will be the atom **community_not_activated**.

`permission_error(create, netobject, Action)`
The predicate was called after the **community_shut/1** call (but before its completion). **ErrInfo-Other** will be the atom **community_is_closing**.

`instantiation_error`
Action is a variable or **ConfigOptionLL** or one member of this list is a variable.

`type_error(atom, Action)`
Action is neither a variable nor an atom.

`domain_error(netconfig_action, Action)`
Action is an atom, but not a valid network configuration action.

`type_error(List, ConfigOptionLL)`
` Either `**`ConfigOptionLL`**` or one of its members is not a`
`   list.domain_error(net_config_option, Opt)`
Opt, one of the options contained inside **ConfigOptionLL** is not a valid net config option for **Action**..

`domain_error(explicit_netobj_name, Name)`
Name given in a **name(Name)** option is not allowed as a user-specified name for a network object. It is **nil** or begins with the reserved character (‘**$**’), or its length exceeds the maximum allowed for network object names (31).

`permission_error(create, netobj_name, Name)`
Name, given in a **name(Name)** option is already the name of an existing partner. The **ErrInfo-Other** will be the atom **partner**.

`existence_error(nethost, Host)`
Host, given in a **hostname(Host)** option is unknown.

`consistency_error(selfhost, self + Host)`
Host, given in a **hostname(Host)** option for the self partner is not valid for our host.

`consistency_error(selfhost, self + mediator)`
mediator(M) option, which qualifies the partner as foreign, is invalid together with the option **self**, which denotes ourselves (the application program running that executed the operation).

`consistency_error(netaddr, Protocol + Kind)`
Protocol, given in a **protocol(Protocol)** option, and **Kind**, the kind of the mediator specified in the **mediator(M)** option, are contradicting. **ErrInfo-Other** will be the atom **protocol_is_invalid_for_this_mediator**.

`consistency_error(netaddr, Protocol + prolog)`
Protocol, given in a **protocol(Protocol)** option, is not accepted for normal (prolog) partners. (At present only tcp can be used in communication with prolog partners.)

`domain_error(ip_addr, Addr)`
Addr, given in an **ip_addr(Addr)** option is not a valid TCP/IP address.

`consistency_error(netaddr, Host + Addr)`
Host given in a **hostname(Host)** option, and **Addr** given in a **ip_addr(Addr)** option, are contradicting (**Addr** is not recognized as one of the registered TCP/IP addresses of **Host**).

`existence_error(netservice, Service)`
Service, given in a **service(Service)** option is unknown.

`domain_error(not_less_than_zero, TcpPort)`
TcpPort, given in a **ip_port(TcpPort)** option is an integer less than zero

`domain_error(`**`ip_port`**`, TcpPort)`
TcpPort given in a **ip_port(TcpPort)** option exceeds the limit for valid port numbers.

`permission_error(modify, netobject, self)`
The reconfiguration of the self partner is requested.

`existence_error(netobject, Name)`
Name, given in a **name(Name)** option for **remove** action is unknown as partner.

community_shut/1

Description

`community_shut(Mode)`

The predicate shuts down networking. **Mode** can be **`force`** or **`graceful`**. Forced shutdown means immediate break of all network links and destruction of the community. In case of graceful shutdown all messages already sent to the network are delivered, then all connection, partner and port objects are disconnected (closed). The community is destroyed when all network activity is finished.

After completion of the graceful shutdown process the program is optionally notified by an alert. It is not allowed to shut the community gracefully twice.

For more details see: 4.5

Template and modes

`community_shut(+net_close_mode_option)`

Examples

`community_shut(force).`
Closes forcibly the community breaking all connections to all partners.

Errors

`system_error`
The community is not initialized. **ErrInfo-Other** will be the atom **community_not_initialized**.

`instantiation_error`
Mode is a variable.

`type_error(atom, Mode)`
Mode is neither a variable nor an atom.

`domain_error(net_close_mode_option, Mode)`
Mode is an atom, but not a valid close mode.

`permission_error(remove, netobject, community)`
Graceful shutdown for the community is already in progress The **ErrInfo-Other** will be the list **[community, repeated]**.

mediator_create/3
mediator_create/4
mediator_create/5

Description

```
mediator_create(Name, Kind, Dock)
mediator_create(Name, Kind, Dock, Flag1)
mediator_create(Name, Kind, Dock, Flag1, Flag2)
```

Create a new mediator of kind **Kind**, with name **Name**, which will perform data and protocol conversion in the process of communicating with a foreign partner when it is attached to the corresponding partner object (when the partner is created). If **Name** is a variable a new unique mediator name atom is generated by the system and it is unified with the variable. The newly created mediator will be *free* until it is attached to a partner.

Kind specifies the kind of the mediator required. The argument must be a valid kind option atom (at present either **`hnms`** or **`ascii`**).

Dock specifies the dock object to be attached to the new mediator for receiving messages sent by the foreign partner when fully operational. It can be given as **`nil`**, in which case the mediator will silently discard incoming messages instead of passing them to the application. Otherwise it must be the name of a free dock object.

Flag1 and **Flag2** supply values for the corresponding attributes (flag1 and flag2, respectively) of the newly created mediator. The interpretation of these flags depends on **Kind**. The default value for both is the empty string (`''`). These arguments can be specified only when the corresponding mediator accepts the respective flag.

Template and modes

```
mediator_create(?netobj_name_atom, +mediator_kind_option,
                +atom)
mediator_create(?netobj_name_atom, +mediator_kind_option,
                +atom, +atom)
mediator_create(?netobj_name_atom, +mediator_kind_option,
                +atom, +atom, +atom)
```

Examples

```
mediator_create(mediator1, ascii, dock1).
```

A free mediator is created for handling message exchange with a foreign partner in plain ascii text. The incoming messages will be directed to dock **`dock1`**.

`mediator_create(net_mediator, hnms, [], '', private).`

A free mediator is created for supporting message exchange with the HNMS server as foreign partner. The mediator will discard messages coming from the HNMS server (probably only the directory service is used, although messages sent to the HNMS server will be passed in the normal way). The targetted HNMS server manages a HNMS community named '**`private`**' (instead of the default 'public'). Attribute flag1 is explicitly set to the (otherwise default) value `''` because the syntax does not allow for specifying flag2 value without flag1.

Errors

`system_error`
The community is not initialized. **ErrInfo-Other** will be the atom **community_not_initialized**.

`instantiation_error`
Kind, **Dock**, **Flag1**, or **Flag2** is variable.

`type_error(atom, Name)`
Name is neither an atom nor a variable.

`domain_error(explicit_netobj_name, Name)`
Name is an atom, but not valid as a user-specified name for a net object (it is **nil**, or begins with '**$**', or its length exceeds the maximum allowed for network object names (31)).

`permission_error(create, netobj_name, Name)`
Name is already the name of an existing mediator. The **ErrInfo-Other** will be the atom **mediator**.

`domain_error(mediator_kind, Kind)`
Kind is an atom, but not a valid mediator kind.

`type_error(atom, Dock)`
Dock is neither a variable nor an atom.

`existence_error(netobject, Dock)`
There is no dock with name **Dock**. **ErrInfo-Other** will be the atom **dock**.

`permission_error(modify, dock, Dock)`
Dock is not free (already attached to a mediator). **ErrInfo-Other** will be the atom **engaged**.

`type_error(atom, Flag1)`
Flag1 is neither an atom nor a variable.

`domain_error(mediator_kind, Flag1)`
Flag1 is an atom, but not valid as a flag1 attribute value for the specified mediator kind.

`permission_error(create, net_attribute, Flag1)`
The mediator of the specified **Kind** does not accept user-defined value for its flag1 attribute. **ErrInfo-Other** will be [mediator, Kind].

`permission_error(create, net_attribute, Flag2)`
The mediator of the specified **Kind** does not accept user-defined value for its flag2 attribute. **ErrInfo-Other** will be [mediator, Kind].

`type_error(atom, Flag2)`
Flag2 is neither an atom nor a variable.

`domain_error(mediator_kind, Flag2)`
Flag2 is an atom, but not valid as a flag2 attribute value for the specified mediator kind.

mediator_close/1

Description

`mediator_close(Name)`

Deletes the mediator with name **Name**. Only a *free* or *zombie* mediator can be closed. Closing a mediator succeeds immediately (synchronously). No alert is sent.

Template and modes

`mediator_close(+atom)`

Examples

`mediator_close(mediator1).`
The mediator is destroyed without sending an alert.

Errors

`system_error`
The community is not initialized. **ErrInfo-Other** will be the atom **community_not_initialized**.

`instantiation_error`
Name is a variable.

`type_error(atom, Name)`
Name is neither a variable nor an atom.

`existence_error(netobject, Name)`
There is no mediator with name **Name**. **ErrInfo-Other** will be the atom **mediator**.

`permission_error(remove, netobject, Name)`
The mediator is not zombie (it is associated with an active dock). The **ErrInfo-Other** will be the list **[mediator, engaged]**.

port_create/2
port_create/3
port_create/4
port_create/5

Description

```
port_create(Name, Channel)

port_create(Name, Channel, Advertised)

port_create(Name, Channel, Advertised, Buffering)

port_create(Name, Channel, Advertised, Buffering,
            EndMarker)
```

Create a new port with name **Name,** which will feed messages into **Channel**. If **Name** is a variable a new unique port name atom is generated by the system and it is unified with the variable. **Advertised** can be **`on`** or **`off`**; it controls the publicity status of the port. Advertised port names are broadcast to partners where they are stored in an attribute value for partner objects (**`port_names`**). Names of non-advertised ports cannot be retrieved by partners directly from this partner attribute; they can be connected only if the partner knows the port name in advance. The default value for **Advertised** is **`off`**.

Buffering specifies the maximum number of unprocessed messages buffered for the port. Value of `0` means that the port will accept a message from a remote connection if the previous one from the same connection (if any) has been consumed. Positive **Buffering** value is an advice to the CS-Prolog system, that it may accept more than one message from a remote connection. The default value for **Buffering** is `0`. Buffering value is stored in the modifiable port attribute **`buffering_limit`**.

EndMarker can be **`on`** or **`off`**. If its value is **`on`**, the system inserts a special message whenever a connection to the port is closed or broken. The format of this message is **`end_of_message_stream(Mode)`** where **`Mode`** is the disconnection mode (network errors imply **`force`**). If the value **`off`** is specified for **EndMarker**, or if this parameter is omitted, no terminating message is inserted.

Template and modes

```
port_create(?netobj_name_atom, +channel)
port_create(?netobj_name_atom, +channel, +option)
port_create(?netobj_name_atom, +channel, +option, +integer)
port_create(?netobj_name_atom, +channel, +option, +integer,
            +option)
```

Examples

`port_create(air_port, ch(air_port)).`

A non-advertised port is created.

`port_create(eric_port, eric_chan, on, 12, on).`

An advertised port is created, which may store up to 12 unprocessed messages. Each time a remote connection to this port is disconnected an end of message stream term will be inserted for the port.

Errors

`system_error`
The community is not initialized. **ErrInfo-Other** will be the atom **community_not_initialized**.

`instantiation_error`
Channel, **Advertised**, **Buffering,** or **EndMarker** is a variable, or **Channel** contains a variable.

`type_error(atom, Name)`
Name is neither an atom nor a variable.

`domain_error(explicit_netobj_name, Name)`
Name is an atom, but not valid as a user-specified name for a net object (it is **nil**, or begins with '**$**', or its length exceeds the maximum allowed for network object names (31)).

`type_error(atom, Advertised)`
Advertised is neither a variable nor an atom.

`domain_error(option, Advertised)`
Advertised is an atom, but not a valid option (**off** or **on**).

`type_error(integer, Buffering)`
Buffering is neither a variable nor an integer.

`domain_error(not_less_than_zero, Buffering)`
Buffering is an integer less than zero.

`type_error(atom, EndMarker)`
EndMarker is neither a variable nor an atom.

`domain_error(option, EndMarker)`
EndMarker is an atom, but not a valid option (**off** or **on**).

`permission_error(create, netobj_name, Name)`
Name is already the name of an existing port. The **ErrInfo-Other** will be the atom **port**.

`domain_error(unique_name, Channel)`
Channel is not a valid unique name.

`permission_error(open, channel, Channel)`
The **Channel** channel is already opened for send. The **ErrInfo-Other** will be the atom **already_open**

`permission_error(open, channel, Channel)`
The **Channel** channel is opened for receive by a connection. The **ErrInfo-Other** will be the atom **already_network**.

port_close/1
port_close/2

Description

`port_close(Name)`

`port_close(Name, Mode)`

Deletes the port with name **Name** in mode **Mode**. In **`graceful`** mode the destruction of the port will be delayed until all connected remote connections are informed and disconnected and all unprocessed messages are received by the program. In **`force`** mode the destruction is immediate, partners are informed, but the unconsumed messages in the port's buffer (and messages on their way on network channels) are discarded. The default close mode is **`graceful`**.

When **port_close/[1,2]** succeeds the attached channel is closed; it becomes available for opening for send by any process. At completion of graceful port closing the system sends an alert informing the program about this event.

It is not allowed to close a port gracefully more than once.

For more details see: 4.5

Template and modes

```
port_close(+atom)
port_close(+atom, +net_close_mode_option)
```

Examples

```
port_close(air_port, force).
```
The port is destroyed without caring for partners and unconsumed messages.

Errors

`system_error`
The community is not initialized. **ErrInfo-Other** will be the atom **community_not_initialized**.

`instantiation_error`
Name or **Mode** is a variable.

`type_error(atom, Name)`
Name is neither a variable nor an atom.

`existence_error(netobject, Name)`
There is no port with name **Name**. **ErrInfo-Other** will be the atom **port**.

`type_error(atom, Mode)`
Mode is neither a variable nor an atom.

`domain_error(net_close_mode_option, Mode)`
Mode is an atom, but not a valid close mode.

`permission_error(remove, netobject, Name)`
Graceful close for port with name **Name** is already in progress The **ErrInfo-Other** will be the list **[port, repeated]**.

connect_to_port/4
connect_to_port/5

Description

`connect_to_port(Name, Partner, Port, Channel)`

`connect_to_port(Name, Partner, Port, Channel, Buffering)`

Create a new connection with name **Name,** which will forward messages fed into **Channel** to the port **Port** of partner **Partner**. If **Name** is a variable the system generates a new unique connection name atom which is unified with **Name**. If **Partner** is in initial state the action is suspended until the net connecting process to the partner completes (and partner's status becomes **`normal`**). The existence of the port is checked on partner's side. If either the partner is or becomes zombie, or the requested port does not exist at the partner, then the new connection's status is set to **`zombie`** and optionally an alert is generated.

Buffering specifies the maximum number of messages stored in the connection object waiting to be sent (the permission comes from the remote port). A value of **`0`** means that the connection will accept only one message and the send operation will be blocked until the addressee on the other side receives the message and acknowledges the receipt (the **send** is synchronized). When positive buffering value is given, the send operation will not block as long as there is space in the message buffer and the number of unsent messages does not exceed **Buffering**. In this case send is not synchronized. **Buffering** value is stored in the modifiable connection attribute **`buffering_limit`**. However, the *zero-ness* of this attribute cannot be changed later, i.e., if it is zero, it cannot be changed to non-zero and similarly non-zero cannot be changed to zero.

The program is optionally informed about success or failure of a **connect_to_port** call by an appropriate alert.

Template and modes

```
connect_to_port(?netobj_name_atom, +atom, +atom, +channel)
connect_to_port(?netobj_name_atom, +atom, +atom, +channel,
                +integer)
```

Examples

`connect_to_port(airlink, near, air_port, chan(airlink)).`
A new connection is created to the remote port **`air_port`** of partner **`near`**. The program can send messages there through the channel **`chan(airlink)`**. The send opeartion will be synchronized with the receiver.

Errors

`system_error`
The community is not initialized. **ErrInfo-Other** will be the atom **community_not_initialized**.

`instantiation_error`
Partner, **Port**, **Channel**, or **Buffering** is a variable, or **Channel** contains a variable.

`type_error(atom, Name)`
Name is neither an atom nor a variable.

`domain_error(explicit_netobj_name, Name)`
Name is an atom, but not valid as a user-specified name for a net object (it is **nil**, or begins with **'$'**, or its length exceeds the maximum allowed for network object names (31)).

`type_error(atom, Partner)`
Partner is neither a variable nor an atom.

`existence_error(netobject, Partner)`
There is no partner with name **Partner**. **ErrInfo-Other** will be the atom **partner**.

`type_error(atom, Port)`
Port is neither a variable nor an atom.

`domain_error(unique_name, Channel)`
Channel is not a valid unique name.

`type_error(integer, Buffering)`
Buffering is neither a variable nor an integer.

`domain_error(not_less_than_zero, Buffering)`
Buffering is an integer less than zero.

`permission_error(create, netobj_name, Name)`
Name is already the name of an existing connection. **ErrInfo-Other** will be the atom **connection**.

`permission_error(open, channel, Channel)`
The channel **Channel** is already opened for receive. **ErrInfo-Other** will be the atom **already_open**

`permission_error(open, channel, Channel)`
The channel **Channel** is opened for send by a port or a dock. **ErrInfo-Other** will be the atom **already_network**.

dock_create/2
dock_create/3
dock_create/4

Description

```
dock_create(Name, Channel)
dock_create(Name, Channel, Buffering)
dock_create(Name, Channel, Buffering, EndMarker)
```

Create a new dock with name **Name,** which will feed messages into **Channel** when it is attached to a mediator attached to a foreign partner. If **Name** is a variable a new unique dock name atom is generated by the system and it is unified with the variable. The newly created dock will be *free* until it is attached to a mediator

Buffering specifies the maximum number of unprocessed messages buffered for the dock. Value of `0` means that the dock will accept a message from a remote connection if the previous one from the same connection (if any) has been consumed. Positive **Buffering** value is an advice to the CS-Prolog system, that it may accept more than one message from a remote connection. The default value for **Buffering** is `0`. Buffering value is stored in the modifiable dock attribute **`buffering_limit`**.

EndMarker can be **`on`** or **`off`**. If its value is **`on`**, the system inserts a special message when the conversation with the foreign partner connected to the dock is closed or broken. The format of this message is **`end_of_message_stream(Mode)`** where **`Mode`** is the disconnection mode (network errors imply **`force`**), or, in the case of a dock attached to an **ascii** mediator, the atom **`end_of_message_stream`**. If the value **`off`** is specified for **EndMarker**, or if this parameter is omitted, no terminating message is inserted.

Template and modes

```
dock_create(?netobj_name_atom, +channel)
dock_create(?netobj_name_atom, +channel, +integer)
dock_create(?netobj_name_atom, +channel, +integer, +option)
```

Examples

```
dock_create(dock1, dch1).
```
A non-buffering dock is created.

```
dock_create(eric_dock, eric_chan, 12, on).
```
A dock is created, which may store up to 12 unprocessed messages. When the conversation with the foreign partner connected to this dock is terminated, an end of message stream term will be inserted for the dock.

Errors

```
system_error
```
The community is not initialized. **ErrInfo-Other** will be the atom **community_not_initialized**.

`instantiation_error`
Channel, **Buffering,** or **EndMarker** is a variable, or **Channel** contains a variable.

`type_error(atom, Name)`
Name is neither an atom nor a variable.

`domain_error(explicit_netobj_name, Name)`
Name is an atom, but not valid as a user-specified name for a net object (it is **nil**, or begins with **'$'**, or its length exceeds the maximum allowed for network object names (31)).

`type_error(integer, Buffering)`
Buffering is neither a variable nor an integer.

`domain_error(not_less_than_zero, Buffering)`
Buffering is an integer less than zero.

`type_error(atom, EndMarker)`
EndMarker is neither a variable nor an atom.

`domain_error(option, EndMarker)`
EndMarker is an atom, but not a valid option (**off** or **on**).

`permission_error(create, netobj_name, Name)`
Name is already the name of an existing dock. The **ErrInfo-Other** will be the atom **dock**.

`domain_error(unique_name, Channel)`
Channel is not a valid unique name.

`permission_error(open, channel, Channel)`
The **Channel** channel is already opened for send. The **ErrInfo-Other** will be the atom **already_open**

`permission_error(open, channel, Channel)`
The **Channel** channel is opened for receive by a connection. The **ErrInfo-Other** will be the atom **already_network**.

dock_close/1

dock_close/2

Description

`dock_close(Name)`

`dock_close(Name, Mode)`

Deletes the dock with name **Name** in mode **Mode**. Only an *unattached* dock can be closed (free, zombie, or the partner with which the dock is associated is zombie). Closing a free or zombie dock succeeds immediately. Otherwise in **`graceful`** mode the destruction of the dock will be delayed until all unprocessed messages are received by the program. In **`force`** mode the destruction is immediate, the unconsumed messages in the dock's buffer (and messages on their way on network channels) are discarded. The default close mode is **`graceful`**.

When **dock_close/1** succeeds the attached channel is closed; it becomes available for opening for send by any process. At completion of graceful dock closing the system sends an alert informing the program about this event.

It is not allowed to close a dock gracefully more than once.

For more details see: 4.5

Template and modes

`dock_close(+atom)`

Examples

`dock_close(dock1).`
The dock is destroyed without sending an alert.

Errors

`system_error`
The community is not initialized. **ErrInfo-Other** will be the atom **community_not_initialized**.

`instantiation_error`
Name is a variable.

`type_error(atom, Name)`
Name is neither a variable nor an atom.

`existence_error(netobject, Name)`
There is no dock with name **Name**. **ErrInfo-Other** will be the atom **dock**.

`permission_error(remove, netobject, Name)`
The dock is not free (it is associated with an active partner **-** through a mediator) The **ErrInfo-Other** will be the list **[dock, engaged]**.

connect_to_dock/4

connect_to_dock/5

Description

```
connect_to_dock(Name, Partner, Dock, Channel)
connect_to_dock(Name, Partner, Dock, Channel, Buffering)
```

Create a new connection with name **Name,** which will forward messages fed into **Channel** to the dock **Dock** of partner **Partner**. If **Name** is a variable the system generates a new unique connection name atom, which is unified with **Name**. If **Partner** is in initial state the action is suspended until the net connecting process to the partner completes (and partner's status becomes **`normal`**). **Dock** can be specified as **`nil`** in which case the default dock of the foreign partner is the target of the connection. The existence of the dock is checked on the partner's side. If either the partner is or becomes zombie, or the requested dock does not exist at the partner, then the new connection's status is set to **`zombie`** and optionally an alert is generated.

Buffering specifies the maximum number of messages stored in the connection object waiting to be sent (the permission comes from the remote dock). A value of **`0`** means that the connection will accept only one message and the send operation will be blocked until the addressee on the other side receives the message and acknowledges the receipt (the send is synchronized). When positive buffering value is given, the send operation will not block as long as there is space in the message buffer and the number of unsent messages does not exceed **Buffering**. In this case send is not synchronized. **Buffering** value is stored in the modifiable connection attribute **`buffering_limit`**. However, the *zero-ness* of this attribute cannot be changed later, i.e., if it is zero, it cannot be changed to non-zero and similarly non-zero cannot be changed to zero.

The program is optionally informed about success or failure of a **connect_to_dock** call by an appropriate alert.

Template and modes

```
connect_to_dock(?netobj_name_atom, +atom, +atom, +channel)
connect_to_dock(?netobj_name_atom, +atom, +atom, +channel,
                +integer)
```

Examples

```
connect_to_dock(CONNAME, stranger, [], chan(1)).
```

A new connection is created to the default dock of foreign partner **`stranger`**. The program can send messages there through the channel **`chan(1)`**. The send opeartion will be synchronized with the receiver. The variable **CONNAME** will be instantiated with the system-generated name of the new connection.

Errors

`system_error`
The community is not initialized. **ErrInfo-Other** will be the atom **community_not_initialized**.

`instantiation_error`
Partner, **Dock**, **Channel**, or **Buffering** is a variable, or **Channel** contains a variable.

`type_error(atom, Name)`
Name is neither an atom nor a variable.

`domain_error(explicit_netobj_name, Name)`
Name is an atom, but not valid as a user-specified name for a net object (it is **nil**, or begins with ‘**$**’, or its length exceeds the maximum allowed for network object names (31)).

`permission_error(create, netobj_name, Name)`
Name is already the name of an existing connection. The **ErrInfo-Other** will be the atom **connection**.

`type_error(atom, Partner)`
Partner is neither a variable nor an atom.

`existence_error(netobject, Partner)`
There is no partner with name **Partner**. **ErrInfo-Other** will be the atom **partner**.

`type_error(atom, Dock)`
Dock is neither a variable nor an atom.

`domain_error(unique_name, Channel)`
Channel is not a valid unique name.

`type_error(integer, Buffering)`
Buffering is neither a variable nor an integer.

`domain_error(not_less_than_zero, Buffering)`
Buffering is an integer less than zero.

`permission_error(create, netobj_name, Name)`
Name is already the name of an existing connection. **ErrInfo-Other** will be the atom **connection**.

`permission_error(open, channel, Channel)`
The channel **Channel** is already opened for receive. **ErrInfo-Other** will be the atom **already_open**

`permission_error(open, channel, Channel)`
The channel **Channel** is opened for send by a port or a dock. **ErrInfo-Other** will be the atom **already_network**.

disconnect/1

disconnect/2

Description

`disconnect(Name)`

`disconnect(Name, Mode)`

Destroy the connection with name **Name** in mode **Mode**. The local channel is closed at once. In **`graceful`** mode the completion of disconnect is delayed until all messages remaining on the connection's buffer are delivered to the receiver. At the end the connected port (of the partner) is informed about disconnection and the connection object is deleted. In **`force`** mode the disconnect is immediate, the remote port is informed, but the pending messages in the connection's buffer (and messages on their way on the network) are discarded. The default disconnect mode is **`graceful`**.

When a graceful **disconnect/[1,2]** succeeds the system optionally sends an alert informing the program. It is not allowed to disconnect a particular connection gracefully more than once.

For more details see: 4.5

Template and modes

```
disconnect(+atom)
disconnect(+atom, +net_close_mode_option)
```

Examples

`disconnect(airlink, force).`
The connection is disconnected without delay. Messages waiting to be delivered are discarded.

Errors

`system_error`
The community is not initialized. **ErrInfo-Other** will be the atom **community_not_initialized**.

`instantiation_error`
Name or **Mode** is a variable.

`type_error(atom, Name)`
Name is neither a variable nor an atom.

`existence_error(netobject, Name)`
There is no connection with name **Name**. **ErrInfo-Other** will be the atom **connection**.

`type_error(atom, Mode)`
Mode is neither a variable nor an atom.

`domain_error(net_close_mode_option, Mode)`
Mode is an atom, but not a valid close mode.

`permission_error(remove, netobject, Name)`
Graceful disconnect for the connection with name **Name** is already in progress The **ErrInfo-Other** will be the list **[connection, repeated]**.

community_current_attribute/3
partner_current_attribute/3
port_current_attribute/3
dock_current_attribute/3
connection_current_attribute/3
mediator_current_attribute/3

Description

```
community_current_attribute(ObjName, AttrName, Value)
partner_current_attribute(ObjName, AttrName, Value)
port_current_attribute(ObjName, AttrName, Value)
dock_current_attribute(ObjName, AttrName, Value)
connection_current_attribute(ObjName, AttrName, Value)
mediator_current_attribute(ObjName, AttrName, Value)
```

These predicates serve for retrieving or checking attribute values of network objects. Each argument can be a variable. If **ObjName** is not a variable then it has to be the name of a network object of appropriate type (community, partner, port, or connection, respectively). If **AttrName** is not a variable then it has to be a valid attribute name for the object type. If **AttrName** and **Value** both are instantiated, then **Value** has to be a value valid for this attribute. The network attributes are listed in section 5.

These predicates generate all **O**, **A**, **V** triplets that are object names, attribute names for the given object type and their associated values, and unify **ObjName** with **O**, **AttrName** with **A** and **Value** with **V**. The predicates are resatisfiable, on backtrack they unify all **O**, **A**, **V** triplets with **ObjName**, **AttrName** and **Value**. The used set of **O**, **A**, **V** values is frozen in the moment of the call. So if between two succeedings of the predicates an attribute value is modified, this change does not appear in the result.

At present the name of the (single) community is a fixed atom: **`community`**.

Template and modes

```
community_current_attribute(?atom, ?attr_name, ?attr_value)
partner_current_attribute(?atom, ?attr_name, ?attr_value)
port_current_attribute(?atom, ?attr_name, ?attr_value)
dock_current_attribute(?atom, ?attr_name, ?attr_value)
connection_current_attribute(?atom, ?attr_name, ?attr_value)
mediator_current_attribute(?atom, ?attr_name, ?attr_value)
```

Examples

```
port_current_attribute(X, advertised, on).
```

Unifies **X** with an advertised port name. On backtrack all such port names will be enumerated.

```
partner_current_attribute(near, port_names, L).
```

Unifies **L** with the list of port names advertised by partner **near**.

```
partner_current_attribute(S, self, true),
       ( partner_current_attribute(S, X, Y),
                    format('[~a~25|~q~n', [X,Y]), fail;
         true
       ).
```

Writes out all attributes and attribute values for the self partner.

Errors

`system_error`
The community is not initialized. **ErrInfo-Other** will be the atom **community_not_initialized**.

`type_error(atom, ObjName)`
ObjName is neither a variable nor an atom.

`existence_error(netobject, ObjName)`
ObjName is instantiated and there is no network object of the pertinent type with name **ObjName**. **ErrInfo-Other** will be the atom representing the object type.

`domain_error(net_attribute, AttrName)`
AttrName is instantiated but it is not a valid attribute name for the given object type.

`domain_error(net_attr_value, AttrName+Value)`
AttrName and **Value** are instantiated but **Value** is not a proper attribute value for **AttrName**.

community_current_attribute/4
partner_current_attribute/4
port_current_attribute/4
dock_current_attribute/4
connection_current_attribute/4
mediator_current_attribute/4

Description

```
community_current_attribute(ObjName, AttrName, Value,
                            Mode)
```

```
partner_current_attribute(ObjName, AttrName, Value, Mode)
```

```
port_current_attribute(ObjName, AttrName, Value, Mode)
```

```
dock_current_attribute(ObjName, AttrName, Value, Mode)
```

```
connection_current_attribute(ObjName, AttrName, Value,
                             Mode)
```

```
mediator_current_attribute(ObjName, AttrName, Value,
                             Mode)
```

These predicates are generalizations of predicates having the same name and arity of 3. Each of the first three arguments plays the same role as in the corresponding predicate of arity 3. **Mode** can be **`conditional`** or **`unconditional`**. Unconditional current attribute retrieval has the same effect as described for predicates with 3 arguments. In conditional mode, however, if either the network is uninitialized or **ObjName** is not variable but there is no object with name **ObjName**, then these predicates simply fail instead of signaling an exception as their counterparts do.

Template and modes

```
community_current_attribute(?atom, ?attr_name,
                            ?attr_value, +predicate_mode)
partner_current_attribute(?atom, ?attr_name,
                          ?attr_value, +predicate_mode
port_current_attribute(?atom, ?attr_name,
                       ?attr_value, +predicate_mode
dock_current_attribute(?atom, ?attr_name,
                       ?attr_value, +predicate_mode
connection_current_attribute(?atom, ?attr_name,
                            ?attr_value, +predicate_mode)
mediator_current_attribute(?atom, ?attr_name,
                            ?attr_value, +predicate_mode)
```

Examples

`port_current_attribute(air_port, status, S, conditional).`
Unifies **`S`** with the status of the given port if the port exists. Otherwise the predicate fails.

Errors

`system_error`
Mode is **unconditional** and the community is not initialized. **ErrInfo-Other** will be the atom **community_not_initialized**.

`type_error(atom, ObjName)`
ObjName is neither a variable nor an atom.

`existence_error(netobject, ObjName)`
Mode is **unconditional**, **ObjName** is instantiated and there is no network object of the pertinent type with name **ObjName**. **ErrInfo-Other** will be the atom representing the object type.

`domain_error(net_attribute, AttrName)`
AttrName is instantiated but it is not a proper attribute name for the given object type.

`domain_error(net_attr_value, AttrName+Value)`
AttrName and **Value** are instantiated but **Value** is not a proper attribute value for **AttrName**.

`domain_error(not_less_than_zero, Value)`
Value is a negative integer but for the associated attribute only non-negative values are valid.

community_set_attribute/3

partner_set_attribute/3

port_set_attribute/3

dock_set_attribute/3

connection_set_attribute/3 mediator_set_attribute/3

Description

```
community_set_attribute(ObjName, AttrName, Value)
partner_set_attribute(ObjName, AttrName, Value)
port_set_attribute(ObjName, AttrName, Value)
dock_set_attribute(ObjName, AttrName, Value)
connection_set_attribute(ObjName, AttrName, Value)
mediator_set_attribute(ObjName, AttrName, Value)
```

These predicates change the value of a modifiable attribute of a network object (see section 5).

Template and modes

```
community_set_attribute(+atom, +attr_name, +attr_value)
partner_current_attribute(+atom, +attr_name, +attr_value)
port_current_attribute(+atom, +attr_name, +attr_value)
connection_current_attribute(+atom, +attr_name, +attr_value)
```

Examples

```
port_set_attribute(air_port, advertised, off).
```

The publicity status of **`air_port`** is set to **`off`**. It will be removed from the corresponding partner objects' **`port_names`** attribute value at those remote communities where our application is configured as partner.

Errors

`system_error`
The community is not initialized. **ErrInfo-Other** will be the atom **community_not_initialized**.

`instantiation_error`
ObjName, **AttrName,** or **Value** is a variable.

`type_error(atom, ObjName)`
ObjName is neither a variable nor an atom.

`existence_error(netobject, ObjName)`
There is no network object of the specified type with name **ObjName**. **ErrInfo-Other** will be the atom representing the object type.

`domain_error(net_attribute, AttrName)`
AttrName is not a proper attribute name for the given object type.

`domain_error(net_attr_value, AttrName+Value)`
Value is not a proper attribute value for **AttrName**.

`domain_error(not_less_than_zero, Value)`
Value is a negative integer but for the associated attribute only non-negative values are valid.

`permission_error(modify, net_attribute, AttrName)`
AttrName is a read only attribute.

`permission_error(modify, net_buffering_limit, Value)`
Value is an illegal change of the buffering limit attribute of a connection. (Non-zero values cannot be changed to zero, and zero value cannot be changed to non-zero.)

ask_manager/3

Description

```
ask_manager(ObjName, AttrName, Value)
```

The purpose of this predicate is to obtain information from the network directory service (at present provided only by *the* HNMS partner) about peer applications connected to the same subnetwork manager. The relevant data about each such application is stored in a corresponding instance of a special object type named **NetPeer**. These objects are not (and cannot be) created by the program; they are maintained by the system. Individual **NetPeer** instances appear and disappear as the configuration of the HNMS community changes. The name of the objects is always system-generated, and has no inherent meaning. It can be used only for associating the individual attribute values for a particular peer application with each other.

Because of the ephemeral nature of the **NetPeer** objects, the **ask_manager/3** predicate works in the same manner as the conditional mode querying predicates do, i.e., it simply fails if the object explicitly specified by the (non-variable) **ObjName** argument does not exist. Except for the assumed conditional mode the predicate behaves exactly as any other from the querying predicates group.

Each argument can be a variable. If **ObjName** is not a variable then it has to be the name of a **NetPeer** object instance (obtained from a previous **ask_manager** query or an alert). If **AttrName** is not a variable then it has to be a valid attribute name for the **NetPeer** object type. If **AttrName** and **Value** both are instantiated, then **Value** has to be a value valid for this attribute. The **NetPeer** attributes are listed in section 5.

The predicate generates all **O**, **A**, **V** triplets that are object names, attribute names for the **NetPeer** instances and their associated values, and unify **ObjName** with **O**, **AttrName** with **A** and **Value** with **V**. The predicate is resatisfiable, on backtrack it unifies all **O**, **A**, **V** triplets with **ObjName**, **AttrName** and **Value**. The used set of **O**, **A**, **V** values is frozen in the moment of the call. So if between two succeedings of the predicate an attribute value is modified, this change does not appear in the result.

Note that **NetPeer** instances may exist only when the HNMS partner has normal status.

Template and modes

```
ask_manager(?atom, ?attr_name, ?attr_value)
```

Examples

```
ask_manager(X, host_name, central).
```

Unifies **X** with the system-generated object name of a **NetPeer** object describing a peer application (intelligent agent) running on the host named **`central`**, if the HNMS partner is running, is connected to the HNMS server and has information about such a peer. Othewise the predicat fails. On backtrack all such instances will be enumerated.

```
ask_manager(O, ip_addr, A).
```

Unifies **`A`** with the internet address of a peer application, **`O`** with the system-assigned name of the **NetPeer** object instance describing that peer, if such a peer is known. On backtrack all such addresses will be enumareted.

```
ask_manager(O, host_name, central),
      (ask_manager(O, port_names, L),
                  ask_manager(O, ip_port, P),
                  format('[~a~25|~q~n', [P,L]), fail;
        true
      ).
```

Writes out the tcp/ip port number and the list of the advertised port names for all peer applications running on the host **`central`**.

Errors

`system_error`
The community is not initialized. **ErrInfo-Other** will be the atom **community_not_initialized**.

`type_error(atom, ObjName)`
ObjName is neither a variable nor an atom.

`domain_error(net_attribute, AttrName)`
AttrName is instantiated but it is not a valid attribute name for the NetPeer object type.

`domain_error(net_attr_value, AttrName+Value)`
AttrName and **Value** are instantiated but **Value** is not a proper attribute value for **AttrName**.

9. Changes in existing predicates

Extending CS-Prolog with network communication required changes in the behavior of the already defined predicates. It also meant introduction of additional functionality accessed by new arguments. (The changes are incorporated in Version 2.1 of the User's Manual).

The following parallel programming built-in predicates are affected:

receive/[2,3]

test_channel/2

channel_list/1

An optional argument of receive can be used to obtain information about the originator of a message received from the network.

The channel state record had to be extended to cover the situation when one end of the channel is used for external communication. (Channel state records are referred to in **test_channel/2** and **channel_list/1**.)

receive/4

Description

```
receive(Channel_name, Variable, Winner_channel, Rem_Conn)
```

The first three arguments of the predicate have the same role as in **receive/3**. The **Rem_Conn** argument provides information about the sender. It has to be an unbound variable. If the transfer was local then this variable is unified with **nil**. Otherwise, when the transfer involved the network, the following list is unified with **Rem_Conn**:

```
      [[ip_addr(Ipaddr), ip_port(Tcp_Port)], Connection_Name]
```

`Ipaddr` and **`Tcp_port`** give the full TCP/IP address of the remote application, and **`Connection_Name`** is the name of the sending remote connection. The net address is returned in **Rem_Conn** because the sender application may be an unsolicited partner having no partner representation. If it is an explicit partner, its name can be retrieved using the **partner_current_attribute** predicate.

Template and modes

```
receive(@chanspec, -term, -channel, -term)
```

Examples

```
receive(ch(air_port), Mess, _, Sender),
                      Sender = [_, airlink].
```

Receives a message and then checks whether it had been sent from a remote connection named **`airlink`**.

```
receive(ch(air_port), Mess, _, Sender),
      Sender = [[ip_addr(IA), ip_port(TP], _],
          partner_current_attribute(near, ip_addr, IA),
          partner_current_attribute(near, ip_port, TP).
```

Receives a message and checks whether it had been sent from a partner named **`near`**.

Errors

(only the new error for the **Rem_Conn** argument)

```
type_error(variable, Rem_Conn)
```

The **Rem_Conn** argument is not a variable.

test_channel/2

Description

`test_channel(Channel_name, Channel_state_record)`

The description of this predicate is the same as in CS-Prolog II User’s Manual with the following extension. The fact that one end of the channel is owned by a network object is reflected in the **Sender_process** or **Receiver_process** members of the **Channel_state_record** argument. These members are unified with the list:

```
      [port, Port_Name]
```

or

```
      [connection, Connection_Name]
```

depending on the network object type the channel is linked to. **`Port_Name`** or **`Connection_Name`** is the name of the network object.

10. Implementation-defined limits and constants

Name of the (unique) community object:	`community`
Maximal number of network connections at any one time:	30
Maximal length of network object names:	31
Default ip_port number:	5130
Default retry delay period:	6000 (hundredth sec's)
Maximal length of a message sent to a foreign partner	16382 (16K - 2) bytes
Maximal summary length of string data for a NetPeer object	370 bytes

11. APPENDIX A — MEDIATORS AVAILABLE

11.1 The *ascii* mediator for plain text communication

This mediator, which is integrated with the network driver, should be used in communicating with server-like partners that

- accept network connection at a pre-defined (*well-known*) port using either TCP/IP or UDP/IP protocol;
- receive and send plain ascii text lines (of limited length).

The mnemonic identifier for creating a mediator of this kind is the atom **ascii**.

The mediator performs the following transformations and poses the following restrictions.

Outgoing messages:

> The term sent to the channel is converted to external representation, as by the 'write' predicate, and a newline character is appended to the result. The resulting character string is then forwarded to the partner.
>
> The length of this string cannot be larger than 16K - 2 bytes.
>
> Note: If there are embedded newline characters inside the string, the partner probably will handle each part as a separate line.
>
> As for message synchronization, a message sent is considered as accepted by the remote partner at the moment when the mediator passed the last character of it to the network (maybe after a period of the message staying buffered).

Incoming messages:

> The character stream arriving from the partner is divided into individual lines. If a line is longer than the maximal atom length allowed in CSP (4095 characters), than that line is further split into parts (so that each part except perhaps the last one has the maximal length allowed). The 'receive' predicates get the next consecutive part converted to a prolog atom (without the terminating newline character).
>
> When the conversation is terminated gracefully (because of the partner's removal, or because of networking error, or maybe the remote partner decides to close the communication), optionally an **`end_of_message_stream`** atom is inserted as the last message to be received, under control of the **EndMarker** attribute specified or assumed when the receiving dock was created.

11.2 The *hnms* mediator for communication with the HNMS server

This mediator has a separate executable component named ***csphnmsd***.

The mnemonic identifier for creating a mediator of this kind is the atom **hnms**.

In contrast to the rather general ascii mediator, the hnms mediator is quite specific, with a specific set of rules. It provides two distinct services.

One is the directory service. The mediator — after it had contacted the HNMS server — keeps track of all peer CSP applications connected to the same HNMS server and makes the relevant data available via the **ask_manager/3** predicate.

The other service is user controlled message exchange with the HNMS server, consisting of sending HNMS-specific commands and receiving data sent by the HNMS server as instructed by these commands (subscription results and replies to simple queries).

The format of these messages is regulated by the requirements and capabilities of the HNMS server. The additional restriction posed by the mediator is that the external representation of the Prolog terms constituting the 'query' messages must not be longer than 16K - 2 bytes.

When the HNMS partner is created successfully, it launches the so called hnms driver (as a separate unix process) and passes it some parameters. Among these there are the value of the **hostname** attribute of the partner object and the **flag1** and **flag2** attributes of the attached mediator. The driver then tries to contact the HNMS server, using the received **hostname** value to locate the host machine where the server should be running, and the **flag2** value for the name of HNMS community supported by that server, or the normal default value for either of these if the corresponding item is not explicitly specified. The default community name is assumed also when **flag2** is explicitly given as the empty string.

In the present implementation the actual network address of the HNMS server is not made available for the application program. Instead of the actual address fictive values are used for identification purposes (e.g., as partner attribute values): **`'0.0.0.0'`** for **ip_addr** and **`65535`** as (UDP) **ip_port**, which are otherwise invalid.

When the contact with the server is established, the driver begins to collect information about the subnetwork managed by the server (from messages received from the server). The value of the **flag1** attribute received from the parent process controls the eventual automatic subscriptions requested when an object of the appropriate type is recognized.

At present this flag can be an atom (string) composed of the letters **`a`**, **`i`**, **`p`**, and **`r`**. Each letter can be included at most once. (Examples of valid values: **`''`**, **`'pi'`**, **`'ipra'`**.)

The effect of the individual letters included is the same as that of the corresponding command line switch specified for the **hnmstool** program.

The interface for sending requests to, and receiving responses from, the driver is modeled after the **hnmstool** command interface, but only a subset of the functions is implemented.

Both requests and responses are handled as Prolog terms of specific structure, sent and received as CSP messages. For the sake of compactness, a semi-formal syntax description is used to define the simple 'language' that generates/recognizes these terms.

11.2.1 Syntax notation

```
<REQUEST> ::= list                                |
              show(<ObjDesc>)                     |
              translate(<OidIn>)                  |
              subscribe(<ObjDesc>, <VarSubList>)  |
              unsubscribe(<ObjDesc>)              |
              get(<ObjDesc>, <VarDesc>)           |
              getnext(<ObjDesc>, <VarDesc>)       |
              walk(<ObjDesc>, <VarDesc>)


<REPLY>   ::= response(<ObjDesc>, <ValueList>)            |
              showing(<ObjDesc>, <ValueList>, <ComplInd>) |
              translation(<OidIn>, <OdeOut>, <TypeCode>)  |
              objects(<ObjDescList>, ComplInd)            |
              rejected(<REQUEST>, <ErrDiag>)              |
              overrun(<Ts1>, <Ts2>, <Count>)              |
              end_of_message_stream(<F>)


<VarSubList>   ::= [<VarSubItem>, ... ]
<ValueList>    ::= [<ValueItem>, ... ]
<ObjDescList>  ::= [<ObjDesc>, ... ]

<VarSubItem>   ::= [<VarDesc>, <Interval>]
<ValueItem>    ::= [<VarDesc>, <TypeCode>, <Value>, <Ts>]

<ObjDesc>      ::= <class> : <object_name>

<VarDesc>      ::= <MIB_VARIABLE>
<OidIn>        ::= <MIB_VARIABLE>
<OdeOut>       ::= <MIB_VARIABLE>

<Ts>           ::= <TIMESTAMP>
<Ts1>          ::= <TIMESTAMP>
<Ts2>          ::= <TIMESTAMP>

<ComplInd>     ::= complete | continuing | last
<F>            ::= force | graceful

<Value>        ::= ATOM | INTEGER | FLOAT

<class>        ::= ATOM
<object_name>  ::= ATOM
<MIB_VARIABLE> ::= ATOM
<ErrDiag>      ::= ATOM
<TypeCode>     ::= INTEGER
<Count>        ::= INTEGER
<Interval>     ::= INTEGER | FLOAT
<TIMESTAMP>    ::= FLOAT
```

11.2.2 Explanation of the syntax

ATOM, **INTEGER**, and **FLOAT** denote Prolog atom, integer, and float number, respectively.

<TIMESTAMP> is the CS-Prolog II compact representation of a date/time value. It is a floating point value showing the number of days between the designated time-point and the base date fixed for CS-Prolog. (More detailed description of this data/time representation can be found in the accompanying volume about the data base interface extension to CS-Prolog II)

The shorthand notation

```
                [<Item>, ... ]
```

stands for non-empty prolog lists.

<Interval> is the interval **in hundredth of seconds** specified for a periodic subscription. In this case we differ from **hnmstool** (and from the HNMS specification in general) where interval is given in seconds. In CS-Prolog II, however, time intervals are uniformly expressed in hunsecs, so we adhere to this convention. In actual use the specified interval is rounded up to the nearest second.

<MIB_VARIABLE> is any acceptable representation of a MIB variable as a Prolog atom. (Optional symbolic prefix followed by an optional dot-separated node-list, where both parts cannot be empty.) HNMS always returns the representation with the shortest node-list based on its stored MIB-subtree.

<Value_item> represents MIB variable values as a quadruple consisting of the name of the variable (Ode), its defined type, the associated value and the timestamp when this value had been registered.

The **<TypeCode>** mapping and the Prolog representation of the associated value is the following:

TypeCode	HNMS mnemonic	Prolog term
1	MIB_integer	**INTEGER** or **FLOAT**
2	MIB_enum	**INTEGER**
3	MIB_timeticks	**INTEGER** or **FLOAT**
4	MIB_counter	**INTEGER** or **FLOAT** (positive)
5	MIB_gauge	**INTEGER** or **FLOAT**
6	MIB_ipaddr	**ATOM** (standard Internet dot address)
7	MIB_octetstring	**ATOM**
8	MIB_displaystring	**ATOM**
9	MIB_oid	**ATOM** (**<MIB_VARIABLE>**)
10	MIB_null	**ATOM** ('')
11	MIB_sequence	(Not returned)
12	MIB_regpoint	(Not returned)
13	MIB_other	(Not returned — unknown in HNMS)

In four cases above 'INTEGER or FLOAT' is indicated as the Prolog representation. This means that the term type depends on the actual value. If the value is in the CSP integer range, then it is returned as CSP integer, otherwise as CSP float.

For more exact details on MIB, **hnmstool**, e.t.c., see the HNMS system documentation.

11.2.3 Semantics

The application program may send **`<REQUEST>`** terms to the hnms driver using **send/2**.

The driver interprets each request and either accepts or rejects it. If the request is accepted, there are again two possibilities:

If the driver can handle the request alone, then an appropriate **`<RESPONSE>`** is composed and sent back immediately to the receiving dock.

Otherwise, if the HNMS server is involved, the request is translated into a HNMP message and that massage is forwarded to the HNMS server. The server processes the request and eventually will send some HNMP messages back that can be considered as ‘reply’. Replies can be related to requests only by their content. The hnms driver translates each such HNMP message into a **`<REPLY>`** term and sends it to the receiving dock.

Two queries (**`list`** and **`show`**) can produce very large replies, which are larger than any message received from the HNMS server (they are evaluated locally). If this would be the case, the response to the query is returned in several parts. The last argument in both is a completeness indicator (**`<ComplInd>`**). The value **`complete`** in this position indicates that the whole response is contained in this one message; **`continuing`** indicates a message from a sequence of a multipart response which is not the last one; **`last`** indicates the last message from such a sequence.

The normal correspondence between **`<REQUEST>`** and **`<REPLY>`** is the following:

`<REQUEST>` functor name	**`<REPLY>`** functor name
`list`	**`objects`**
`show`	**`showing`**
`translate`	**`translation`**
`subscribe`	**`response`** (periodically or at change)
`unsubscribe`	N/A
`get`	**`response`**
`getnext`	**`response`**
`walk`	**`response`**

where the **`<REQUEST>`** functor names are the same as the respective HNMP function names.

Special messages received as **`<REPLY>`**:

1. If the hnms driver has any problem in translating the **`<REQUEST>`**, it returns a **`rejected`** reply, composed of the original request and a diagnostic term **`<ErrDiag>`**.

   At present, **`<ErrDiag>`** is a simple atom; later on this can be elaborated in more detail. The defined values are summarized in the following table:

`<ErrDiag>` value	Reason
`unknown_request`	Unrecognized main functor of **`<REQUEST>`**
`invalid_arity`	Improper number of arguments in **`<REQUEST>`**
`not_ground_term`	Uninstantiated variable in **`<REQUEST>`**
`type_mismatch`	Violation of the syntax rules for atomic type
`invalid_subterm`	Non-list subterm where list is expected, or **`<ObjDesc>`** is not a ':/2' structure
`invalid_sublist`	Invalid member count in fixed-length sublist, or empty **`<VarSubList>`**
`unknown_object`	No object for **`<ObjDesc>`** found (might have been deleted)

`<ErrDiag>` value	Reason
`unknown_variable_oid`	**`<VarDesc>`** is invalid
`value_error`	Numeric value is out of range (**`<Interval>`** converted to seconds is negative or larger than **`UINT_MAX`**)

2. If the application does not consume the responses quickly enough, the internal buffers of the hnms driver will be filled, and some responses produced in this state will be discarded. The driver keeps track of the number of discarded messages and retains the timestamp of the first and last such response. When 'enough' buffer space becomes available again, the admission of new responses is resumed, and an **`overrun`** message is inserted at the place of the discarded ones in order to let the application know about the situation. The overrun message contains the count and the timestamps mentioned.

3. 3. If the dock associated (indirectly, via the mediator) with the HNMS partner specifies **`EndMarker`** insertion, then the customary **`end_of_message_stream(F)`** term is appended to the stream of responses when the HNMS partner is closed.

11.2.4 Notes

1. HNMS silently discards any message which is larger than its internal output buffer after transformation (16K).
2. Subscription is not incremental, in spite of the specification. A new subscription for an object replaces the older one. This relates to the automatic subscription, too.
3. HNMS message passing is not order-preserving. This means that you should not make any assumptions about the order of expected replies based on the order of the respective requests.
   Three queries (**`translate`**, **`list`**, **`show`**), however, are evaluated locally within the driver and they are order preserving. Also if **`list`** or **`show`** produces multi-part reply, the parts are received without any intervening reply of other kind. (But parts can be lost due to overrun.)
4. In the case when multi-part response is possible, splitting occurs much earlier than strictly necessary in order to avoid very large terms.
5. There is a conflict in the interpretation of the auto-subscribe flags when used together: **`'a'`** subscribes to change; **`'i'`** replaces this by interval subscription (for the MIB variable hnmsObjReachStatus.0).
6. If there are multiple entries in a subscription list for the same MIB variable, then the first one takes effect; the rest are ignored.
7. Flow control: There are three levels of saturation: normal, high, congested.
   In normal state everything goes smoothly; in high level saturation state acknowledgments to received requests are held back, but responses are still produced. In congested state acknowledgments are held back, too, and responses are discarded. When the saturation

level returns to normal from high, withheld acknowledgments are released. If there was a congested state in the meantime, then an overrun report is inserted into the output stream (at the place where the discarded messages should have been enqueued), and response generation is resumed.

8. `rejected` responses are not discarded on saturation.
9. In the present implementation the capability of the directory service is restricted: the total length of the data describing a net-peer must not exceed 384 bytes (the allowed length for string values in HNMS). Of this size 14 bytes are used internally and for the port number, 370 bytes remain for hostname, description, and the advertised portnames (with terminating null character included in each).
10. The directory service relies upon the specific behavior of the **send_relations** and **subscribe_relatons** HNMP functions in HNMS version 2.0 implementation, which differs from the specification. If this anomaly happens to be corrected in future releases, the current hnms driver becomes unusable.

Index of networking built-in predicates