NimBUS

Perl extension for the
NimBUS Message Bus

Nimbus Software as

Contents

1. Introduction 5

2. Concept 5

2.1. Overview 5

3. Nimbus – API 7

4. NimBUS – PDS 11

5. NimBUS – Session 12

6. Nimbus CFG 15

7. Terminology 18

7.1. License System 18

7.2. Nimbus Manager 18

7.3. Nimbus Service Controller 18

7.4. NimBUS domain 18

7.5. Robot 18

7.6. Controller 18

7.7. Spooler 19

7.8. Hub 19

7.9. Probe(s) 19

7.10. NimBUS Address 19

7.11. Message model 19

7.12. Prog. Interfaces 20

7.13. Gateways 20

7.14. Perl 20

Nimbus Software AS

Nimbus Software AS is an independent software consulting and development company located in Oslo, Norway. The company has an extensive knowledge of managing systems, applications and networks in client/server environments. All consultants and system developers are specialists in advanced UNIX and Windows NT technologies and have an in depth understanding of the core technical foundation of these platforms.

Nimbus Software AS
Olaf Helsets vei 6
N-0621 Oslo, Norway
Phone: +47 2262 7160
Fax: +47 2262 7161

e-mail: nimsoft@nimsoft.no
http://www.nimsoft.no

Nimbus Design Strategies

Based on years of development experience and installations of systems management solutions world-wide, spanning global enterprises to small and medium sized companies, we have focused on the following areas when developing the Nimbus Technology:

- Minimum impacts on system resources like CPU, disk and memory.
- Intelligent message handling to avoid unnecessary networks traffic, i.e. message suppression.
- 100% guaranteed message transfer - no lost messages. Lost messages is typically a problem using unreliable network management protocols like SNMP or in situations where there is network failure.
- Secure message transfer between Robot and receiver by use of cryptographic tools like kerberos or other proprietary protocols.
- The ability to communicate across Firewalls.
- Support for dialled-up ISDN communication.
- Fast and easy deployment, providing rapid return on investment.
- Preserves technology investments by leveraging existing enterprise management solutions as well as integrating existing in-house developed management solutions.
- Scalability, both horizontally and vertically.
- Ease of customisation and development of in-house Robot Probes without having to learn complex, inadequate and proprietary languages.

1. Introduction

Nimbus Software AS has designed the message-bus (NimBUS) to be as open as possible with respect to the various (and specific) user needs. In order to be able to interface the NimBUS, extensions to existing programming environments are available. The following programming languages/environments are currently supported:

- Perl5 (WIN32,UNIX)
- C/C++ using dynamic shared libraries (or as a COM object for WIN32)
- Visual Basic or VB Scripts

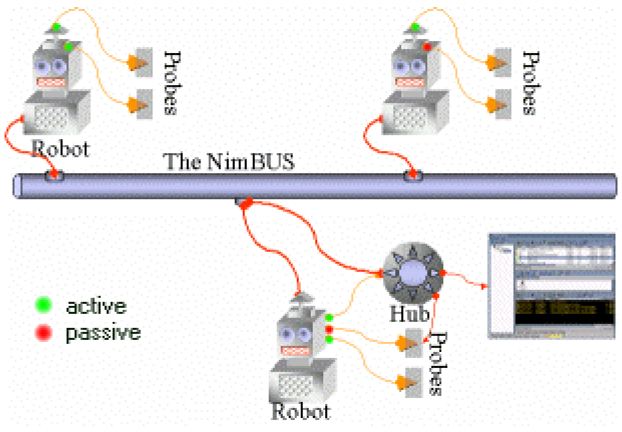
This document describes how you can build your own NimBUS Probes using Perl.

2. Concept

This section describes the NimBUS Concept.

2.1. Overview

The NimBUS



Probe
A program designed specifically towards a certain task, such as monitoring an application, a filesystem, a database etc. or by providing services as a traditional server. The probe may be designed to behave as a *timed* probe by running once and terminating upon completion of the task, or it may be designed as *daemon* probe by running constantly with some kind of waiting mechanism built into it. The *daemon* probes are monitored by the robot controller, and is restarted if it terminates. Probes ca be developed using Perl, Visual Basic or C/C++

Robot
The entry-point for a system (host) into NimBUS. The Robot contains all necessary infrastructure. Its primary task is to maintain and manage as set of probes, and to collect messages sent (published) by its clients. The Robot consists of a *controller* and a *spooler*spooler . A Robot will attempt to establish contact with a *hub* during startup-time. The NimBUS robot will automatically detect *hubs* in the network segment, and connect to one of these (unless specifically specified) during its initialisation process.

HUB
Is the central connection point for a set of robots. It receives all messages posted (sent) by any client (usually via the *spooler*) and distributes these messages to a set of subscribers of the publishing-subject. It keeps track of the NimBUS addresses in the domain of hubs, as well as information about all its robots.

3. Nimbus – API

Nimbus::API - Perl extension for the Nimbus Message Bus

SYNOPSIS

use Nimbus::API;

nimInit(iFlag);
nimEnd (iFlag);

```
my($iRet,$szIp,$iPort) = nimGetNameToIp($szName);
my($iRet,$szName)      = nimGetIpToName($szIp,$iPort);

my($szSup)        = nimSuppToStr($bHold,$iNumber,$iSeconds,$szSuppKey);
my($iRet,$szId)   = nimAlarm($iLevel,$szMsg[,$szSub[,$szSup[,$szSrc]]]);
my($iRet,$szId)   = nimPostMessage($szSubject,$iLevel,$szSup ,$udata);
my($iRet,$rdata)  = nimRequest($szAddr,$iPort,$szCmd,$udata [,$iSec]);
my($nims)         = nimSession ($szAddr, $iPort);
                       nimSessionFree ($nims);

my($iRet,$rdata)  = nimSessionRequest ($nims,$szCmd,$udata [,$iSec]);
my($iRet)         = nimSessionSend ($nims,$szCmd,$udata);
my($iRet)         = nimRegisterProbe ($szProbeName,$iPort);
my($iRet)         = nimUnRegisterProbe ($szProbeName);
my($rc,$value)    = nimGetVarInt($symbol);
my($rc)           = nimSetVarInt($symbol,$value);
my($rc,$value)    = nimGetVarStr($symbol);
my($rc)           = nimSetVarStr($symbol,$value);

my($login)        = nimLogin($user,$password);

nimLogSet($szFile,$szPrefix,$iLevel,$iFlags);
nimLogSetLevel($iLevel);
nimLog ($iLevel,$szString);
nimLogClose();

my($cfg)       = cfgOpen ($szFile,$bReadOnly);
my($iRet)      = cfgClose ($cfg);
my($iRet)      = cfgSync ($cfg);
my($list)      = cfgKeyList ($cfg,$szSection);
my($iRet)      = cfgKeyWrite ($cfg,$szSection,$szKey,$szValue);
my($szValue)   = cfgKeyRead ($cfg,$szSection,$szKey);
my($iRet)      = cfgKeyRename ($cfg,$szSection,$szOld,$szNew);
my($iRet)      = cfgKeyDelete ($cfg,$szSection,$szKey);
my($list)      = cfgSectionList ($cfg,$szSection [,$bRecurse]);
my($iRet)      = cfgSectionDelete ($cfg,$szSection);
my($iRet)      = cfgSectionRename ($cfg,$szOld,$szNew);
my($iRet)      = cfgSectionCopy ($cfg,$szFrom,$szTo);
my($list)      = cfgListRead ($cfg,$szSection);
my($iRet)      = cfgListWrite ($cfg,$szSection,$szKey,$list);
my($iRet)      = cslMatchRegExp ($szTargetString, $szMatchExpr);
```

DESCRIPTION

```
This module will wrap the Nimbus Message Bus (NimBUS), easing
the development of probes written in Perl. The functions available in this
module are data manipulating routines (pds - Portable Data Stream) and the
functions for sending an Alarm, posting a message and sending a request
taken from the Nimbus API.

Conventions used when prototyping the functions:

      sz - prefix for string
      i  - prefix for integer (number)
      b  - prefix for boolean (true(1)/false(0))

      iRet is the Return Code (integer)
```

The client functions:

```
       Initialize the NimBUS
       nimInit(iFlag);
       (This method is called upon loading, and is therefore not
necessary.)
```

- Create suppression definition string.

```
my($szSup) = nimSuppToStr($bHold,$iNumber,$iSeconds,$szSuppKey);

$szSup = nimSuppToStr(0,0,0,"FileSystem|$name");
$szSup = nimSuppToStr(1,0,60,"");
```

- Post alarm message

```
my($iRet,$szId) = nimAlarm($iLevel,$szMsg[,$szSub[,$szSup[,$szSrc]]]);

iLevel is the alarm level (see Level constants).
szMsg  is the alarm message.
szSub  is the subsystem identifier eg. 1.2.3 (default: 1.1).
szSup  is the suppression definition (default: none).
szSrc  is the alarm source (default: localhost).
```

- Post user defined message

```
my($iRet,$szId) = nimPostMessage ($szSubject,$iLevel,$szSup ,$udata);

szSubject is the post channel.
iLevel    is the post priority (see Level constants).
szSup     is the suppression definition.
udata     is a PDS record with user-data.
```

- Map NimBUS name to host-ip and port

```
my($iRet,$szIp,$iPort) =  nimGetNameToIp ($szName);
```

szName is the official NimBUS name (of probe, hub,...).

- Map host-ip and port to NimBUS name

```
my($iRet,$szName) =  nimGetIpToName ($szIp,$iPort);
```

```
szIp  is the hostname or host-ip address of target system.
iPort is the port number of the targeted service.
```

- Send request over the NimBUS to a server.

```
my($retdata) = nimRequest ($szAddr, $iPort, $szCmd, $udata, $iSec);
```

```
        szAddr is the host-name or host-ip.
        iPort  is the service port   (see nimGetNameToIp).
        szCmd  is the service command.
        udata  is the PDS record with user data (see pdsCreate).
        iSec   is time in seconds to wait for reply.

        retdata    is the return PDS record.
```

The constants:

```
            Level constants:
            NIML_CLEAR    (0)
            NIML_INFO     (1)
            NIML_WARNING  (2)
            NIML_MINOR    (3)
            NIML_MAJOR    (4)
            NIML_CRITICAL (5)
```

The log functions:

- Initialize the log system.

```
nimLogSet ($szFile, $szPrefix, $iLevel, $iFlags);
```

```
szFile is the logfile name (or "stdout").
szPre  is the message prefix (incase of multiplexed log).
iLevel is the current loglevel (levels <= iLevel are logged).
iFlags is currently not supported...
```

- Set new loglevel

```
nimLogSetLevel ($iLevel);
```

iLevel is the new loglevel.

- Write to log

```
nimLog ($iLevel, $szString);
```

```
iLevel   is the loglevel (0 is system errors).
szString is the message to log.
```

- Close current log.

```
nimLogClose ();
```

The config-file functions:

- Open config-file

```
my($cfg) = cfgOpen ($szFile, $bReadOnly);
```

- Close config-file

```
my($iRet) = cfgClose($cfg);
```

- Synchronize the config-file (buffer) to disk.

```
my($iRet) = cfgSync($cfg);
```

- Write value to config-file.

```
my($iRet) = cfgKeyWrite ($cfg,$szSection,$szKey,$szValue);
```

- Read value from config-file.

```
my($szValue) = cfgKeyRead ($cfg,$szSection,$szKey);
```

- List array of keys in section from config-file.

```
my($list) = cfgKeyList ($cfg,$szSection);
```

- Rename key in section from config-file.
```
my($iRet) = cfgKeyRename ($cfg,$szSection,$szOld,$szNew);
```

- Delete key in section from config-file.
```
my($iRet) = cfgKeyDelete ($cfg,$szSection,$szKey);
```

- Read array of values from config-file.
```
my($fsList) = cfgListRead ($cfg,$szSection);
```

- Write array of values to config-file.
```
my($iRet) = cfgListRead ($cfg,$szSection,$szKeyBody,$List);
```

- Read array of sections in config-file.
```
my($list) = cfgSectionList ($cfg,"setup");
```

- Delete named section
```
my($iRet) = cfgSectionDelete ($cfg,$szSection);
```

- Rename named section
```
my($iRet) = cfgSectionRename ($cfg,$szOld,$szNew);
```

- Copy named section to a new section
```
my($iRet) = cfgSectionCopy ($cfg,$szFrom,$szTo);
```

- Check string with pattern matching or reg.exp string
```
if (cslMatchRegExp("help me now","*me*")) {
       print "Found match...";
}
```

The data manipulation functions:

```
            PDS  *pdsCreate ();
            int   pdsDelete (PDS *pds);
            int   pdsReset (PDS *pds);
            int   pdsRewind (PDS *pds);
            void  pdsDump (PDS *pds);
            int   pdsSearch (PDS *pds, char *key);
            int   pdsPut_INT (PDS *pds, char *key, int i);
            int   pdsPut_PCH (PDS *pds, char *key, char *s);
            int   pdsPut_PDS (PDS *pds, char *key, PDS *p2);
            int   pdsGet_INT (PDS *pds, char *key);
            char *pdsGet_PCH (PDS *pds, char *key);
            PDS  *pdsGet_PDS (PDS *pds, char *key);

            my($rc,$key,$type,$size,$data) = pdsGetNext(PDS *pds);
```

```
AUTHOR
    Nimbus Software AS. mailto:nimsoft@nimsoft.no,
    http://www.nimsoft.no

SEE ALSO
    perl(1).
```

4. NimBUS – PDS

Nimbus::PDS - Object interface wrapping the PDS

SYNOPSIS

```
     use Nimbus::PDS

     my $pds = Nimbus::PDS->new( [$pdsData] );

              $pds->data();
              $pds->dump();
              $pds->rewind();
              $pds->reset();
              $pds->remove($name);
              $pds->string($name,$value);
              $pds->number($name,$value);
              $pds->putString($name,$value);
              $pds->putNumber($name,$value);
              $pds->put ($name,$value [,$type]);
              $pds->putTable ($name,$value [,$type]);
     $value = $pds->getTable ($name [,$type]);
     $value = $pds->get ($name [,$type]);
     $hptr  = $pds->asHash();
```

DESCRIPTION

The PDS object is a class wrapper around the Nimbus::API PDS functions.

CLASS METODS

- get the get method....
- Put The put method....
- Dump The dump method....
- PutTable The putTable method....
- asHash

The asHash method will produce an associative array (hash) by traversing the PDS. If the PDS contains other PDS's, then the hiearchy will be preserved by nesting.

Example:

```
use Nimbus::PDS;
my $pds = Nimbus::PDS->new();
$pds->putString("name","Donald Duck");
$pds->putString("age",60);

my $h = $pds->asHash();
print "name: $h->{name}, age: $h->{age}\n";
```

AUTHOR

Nimbus Software AS.
mailto:nimsoft@nimsoft.no
http://www.nimsoft.no

SEE ALSO

Nimbus::API, perl(1).

5. NimBUS – Session

Nimbus::Session - Object interface ontop of the NimBUS

SYNOPSIS

```
     use Nimbus::Session

     my $nim = Nimbus::Session->new($id, [$session] );

     $nim->subscribe        ($subjects [,hubip [,hubport]]);
     $nim->attach           ($queue [,hubip [,hubport]]);
     $nim->dispatch         ($timeout_ms [,$breakOnEvent]);
     $nim->addCallback      ($command [,$format [,$security_level]]);
     $nim->server           ([$port, [$timeoutCB [,$restartCB]]]);
     $nim->setInfo          ($version,[$company]);
     $nim->setRetryInterval ($intervalAsSeconds);
     $nim->setPostCallback  ($function_name);
```

DESCRIPTION

```
    The Session object is a class wrapper around the Nimbus::API
    module, and raises the abstraction layer from the low-level
    NimBUS API. You may create a server (TCP/IP), that accepts
    commands over the port(s) registered by the $nim->*server*
    method. The command will be dispatched by the command dispatcher
    to the function with the same name as the command. A command
    without a matching function causes an abort situation. Another
    feature of this class is the connection possibilities to a
    NimBUS hub. The functions $nim->*attach* and $nim->*subscribe*
    both connects to the hub and receives postings over the
    *hubpost* function, see the CALLBACKS manpage.
```

```
 subscribe
    The subscribe method....

  attach
    The attach method....

  dispatch
    The dispatch method....

  addCallback
    The addCallback method....

  server
    The server method....

    When called without parameters the constant NIMPORT_ANY will be
    used. An arbitrary port will be tied to the session. The example
    below illustrates a server that responds to various events
    dispatched by NimBUS.

     Example:

     use Nimbus::Session;

     ####################################################
     # Various callbacks...
     ####################################################
     sub testit {
       my ($hMsg,$level,$name,$age) = @_;
       nimLog(1,"(testit) - level: $level, name: $name, age: $age");
       nimSendReply($hMsg);
     }

     sub debug {
       my ($hMsg,$level) = @_;
       imLog(1,"(debug) from $debug to $level");
       nimSendReply($hMsg);
     }

     sub timeout {
       nimLog(1,"(timeout) - got kicked");
     }

     sub restart {
       nimLog(1,"(restart) - got restarted");
     }


####################################################
# MAIN ENTRY
####################################################
     sub testit {

     $sess = Nimbus::Session->new("perl");

     $sess->setInfo("1.0","Nimbus Software as.");

     if ($sess->server (NIMPORT_ANY,\&timeout,\&restart)==0) {
        $sess->addCallback ("testit","level,name,age%d");
        $sess->addCallback ("debug", "level%d");
     }else {
        nimLog(0,"unable to create server session");
     }

     $sess->dispatch();

  getSessionList

    The getSessionList method....
```

CALLBACKS

```
  The "hubpost" callback function synopsis:

    Whenever 'attach' and 'subscribe' sessions are created, the
    postings will be delivered over the hubpost function. It must be
    declared. This example shows a *very* simple callback function
    looks like, it merely dumps (using pdsDump) the user-data block
    of the posting. It would be normal to extract the values from
    the *$udata* parameter, as 'subject'. The parameters passed to
    the *hubpost* function are:
     $hMsg   - Message handle used by nimSendReply.
     $udata  - User data block (PDS).
     $full   - Complete message block, with embedded udata (PDS).

     sub hubpost {
         my ($hMsg,$udata,$full) = @_;

             $subject = pdsGet_PCH($full,"subject");
         print "The user-data posted under subject: $subject\n";
         pdsDump($udata);

         nimSendReply($hMsg);
     }
```

The command callback function synopsis:

    Every command added by the addCallback method requires a
    callback function such as the one defined below. The $par1 to
    $parN are specified by the *format* element in the addCallback
    parameter list. The $hMsg is the message handle, and is required
    by the Nimbus::API::nimSendReply function.

```
     sub <command> {
         my ($hMsg,$par1,...,$parN) = @_;
             nimSendReply($hMsg);
     }

     Example:

     sub debug {
            my ($hMsg,$level) = @_;
            nimLog(1,"(debug) from $debug to $level");
            nimSendReply($hMsg);
            nimLogSetLevel($level);
     }

     $nim->addCallback("debug","level%d");
```

CHANGES
    Please note that the interface has changed for the following
    methods:

```
     new([$sesslist])                         -> new([$id [,$sesslist]])
     server([$port[,$name[,$cpny[,$vers]]]]) ->
server([$port[,$toutCB[,$restCB]]])
```

    The changes may cause problems for scripts using the
    Nimbus::Session module prior to version 1.05

AUTHOR

 Nimbus Software AS.
 mailto:nimsoft@nimsoft.no
 http://www.nimsoft.no
SEE ALSO

6. Nimbus CFG

SYNOPSIS

```
     use Nimbus::CFG

     my $cfg = Nimbus::CFG->new(["my.cfg" [,$hptr]]);

        $cfg->open("my.cfg" [,$hptr]);
        $cfg->getValues($hptr);
        $cfg->getKeys($hptr);
        $cfg->getSections($hptr);
        $cfg->setConverter(\&src [,\&dst]);
        $cfg->debug($boolean);
            $cfg->dump($cfg);
```

DESCRIPTION

The CFG object is a class wrapper around the functions targeted against configuration files. The relevant functions are Nimbus::API::cfg* .

When a new CFG object is constructed the constructor takes one required argument (the configuration filename), and one optional (a 'private' hash). It would be normal to maintain the hash within the CFG object, but some cases could occur where it would be useful to add the configuration data to a private hash. Eg. when 2 or more configuration-files are referenced by one hash!

Lets call this configuration 'test.cfg':

```
 <setup>
   logfile  = stdout
   loglevel = 2
   <names>
      name_0 = luke
      name_1 = leia
      name_2 = r2d2
   </names>
 </setup>
```

The following code will access the values from the setup section:

```
use Nimbus::CFG;
my $cfg = Nimbus::CFG->new("test.cfg");
print "The logfile : $cfg->{setup}->{logfile}\n";
print "The loglevel: $cfg->{setup}->{loglevel}\n";
```

CLASS METHODS

open

The open method takes a filename/path as a required parameter, and a hashptr as an optional parameter. This method is used when a postponed parsing is needed, due to setting eg. the name converter prior to parsing the file. In result it deliveres the same as a 'new' does.

getValues

The getValues method takes a hashptr as its input parameter, and returns an array of the values taken from each key/value pair in the section.

We're using the configuration file 'test.cfg' from the DESCRIPTION. And the following code segment to extract and access the data:

```
use Nimbus::CFG;
use strict;

my $cfg   = Nimbus::CFG->new("test.cfg");
my @names = $cfg->getValues($cfg->{setup}->{names});
```

@names now holds 3 names...

setConverter

The setConverter method takes a reference to a function as parameter. This function is called whenever a new section is parsed. The default converter substitutes every hash(#) in a section name with a slash(/). This is useful when eg. using the slash (/) character as part of the section/key name, such as a filename or equal. Consider the following code and configuration file:

my.cfg:

```
<filesystems>
  <#dev#dsk#c0t3d0s4>
    name = /usr
        high = 99
        low  = 70
  </#dev#dsk#c0t3d0s4>
</filesystems>
```

```
my_wo_converter.pl:
######################################################
# Script using the standard/builtin converter

use Nimbus::CFG;
my $cfg = Nimbus::CFG->new("my.cfg");

my $fs1 = $cfg->{filesystems}->{'/dev/dsk/c0t3d0s4'};

print "filesystem1: $fs1->{name}, high:$fs1->{high} \n";

==> will print 'filesystem1: /usr, high:99'

my_w_private_conv.pl:
######################################################
# Script using a private converter

use Nimbus::CFG;

sub myconv {
  my $s = shift;
  $$s =~ s/\#/\>/g;   # convert from hash(#) to GT(>)
}

my $cfg = Nimbus::CFG->new();

$cfg->setConverter(\&myconv);
$cfg->open("my.cfg");

my $fs1 = $cfg->{filesystems}->{'>dev>dsk>c0t3d0s4'};

print "filesystem1: $fs1->{name}, high:$fs1->{high} \n";

==> will print 'filesystem1: /usr, high:99'
```

AUTHOR

Nimbus Software AS.
mailto:nimsoft@nimsoft.no
http://www.nimsoft.no

SEE ALSO

Nimbus::API, perl(1).

7. Terminology

7.1. License System

The license information consists of the following parameters:

Product	: Name and version of product or component installed
Info	: Description, name of product, licensee etc..
IP	: IP-address of the system. * means a site-license
#	: Number of clients
Expire	: Date of expire
Code	: License key

In order to use the Nimbus system, a valid *License Certificate* including a **License Key** is required.

NOTE: The product comes with a 30 days valid **License Key**.

The license information can be changes from the HUB Configure program.

7.2. Nimbus Manager

The Nimbus Manager is the main application for administration and configuration of all the nimBUS components. The application has a GUI and is run under Windows.

7.3. Nimbus Service Controller

There is one Windows NT Service running called *Nimbus Watcher.*
The service startup parameters are: Login on as 'System account' and Startup type as' Automatic'.

The application called Nimbus Service Controller is a GUI which makes it easy to start and stop this service. An alternative is to use Services from Control Panel.

7.4. NimBUS domain

7.5. Robot

The robot is the clients (and servers) entry-point into the NimBUS system. Its primary task is to maintain and manage as set of probes, and to collect messages published by its clients. The Robot consists of a controller and spooler . A Robot will attempt to establish contact with a hub during startup-time. The NimBUS robot will automatically detect hubs in the network segment, and connect to one of these (unless specifically specified) during its initialization process.

7.6. Controller

The contact point of a robot seen from other NimBUS components, such as the hub and other clients of the NimBUS. It maintains a set of probes that it starts and stops according to a configuration. The probes may be started in a timed fashion or in a standalone mode called daemon. It responds to requests on the tcp/48000 port.

7.7. Spooler

The spooler receives messages published by the probes (clients) and delivers these messages to the hub, unless its configured to spool (hold) the message until a certain criteria is met. The spooler responds to requests on the tcp/48001 port.

7.8. Hub

As the name implies the hub is a connection point of a collection of robots. It receives all messages posted by any client (usually via the spooler) and distributes these messages to a set of subscribers of the publishing-subject. It keeps track of the NimBUS addresses in the domain of hubs, as well as information about all its robots.

7.9. Probe(s)

A probe is a task-oriented program designed specifially towards a certain task, such as monitoring an application, a filesystem, a database etc. or by providing services as a traditional server. The probe may be designed to behave as a *timed* probe by running once and terminating upon completion of the task, or it may be designed as *daemon* probe by running constantly with some kind of waiting mechanism built into it. The *daemon* probes are monitored by the robot controller, and is restarted if it terminates. A probe may be *active* in the sense of registering itself within the NimBUS system, and responding to commands issued by clients understanding the protocol of the probe. Or it may be *passive*, meaning it doesn't make itself available by the NimBUS addressing scheme.

7.10. NimBUS Address

A NimBUS address consists of four basic elements, the *domain, hub, hostname* and *probe*. For example, the address /nimbus/oslo/wscase/nas will resolve into the ip-address of the host *wscase* and the port-number of the probe/service called *nas*.

7.11. Message model

The NimBUS message model are based on the *request/response* and the *publish/subscribe* models. *Request/Response* are the standard ways of communicating over the network. A client issues a request to a server and the server responds to the request. The *publish/subscribe* model is useful when a client wishes to send of some kind of data without a designated receiver. This could be messages containing performance-data, an alert, data to be inserted into some database, or messages targeted for gateway servers. The servers on the other hand merely listens on a one or more specific subjects (registered by the hub), and is notified by events when data is available on the subject.

7.12. Prog. Interfaces

It is possible to interface the NimBUS by using many of the industry-leading programming languages, such as C/C++,Perl,Visual Basic, VB Script and JAVA. The interfaces makes it possible to quickly and seamlessly integrate the NimBUS with existing tools, or by tailoring the ie. surveillance system for your specific needs.

7.13. Gateways

A service (an *active probe*) used to interface other environments, such as other Enterprise management tools, SNMP based management tools, paging systems, mail (SMTP) etc.

7.14. Perl

From the net:

- Perl is an interpreted high-level programming language developed by Larry Wall. According to Larry, he included in Perl all the cool features found in other languages and left out those features that weren't so cool.
- Perl has become the premier scripting language of the Web, as most CGI programs are written in Perl. However, Perl is widely used as a rapid prototyping language and a "glue" language that makes it possible for different systems to work well together.
- Perl is popular with system administrators who use it for an infinite number of automation tasks.
- Perl's roots are in UNIX but you will find Perl on a wide range of computing platforms. Because Perl is an interpreted language, Perl programs are highly portable across systems.
- Finally, Perl is more than a programming language. It is a part of the Internet culture. It is a very creative way of thinking about almost anything.

For more detailed information see:
- http://www.perl.com
- http://www.ActiveState.com