

YAMI Property Server Documentation

Copyright © 2001-2008 Maciej Sobczak

Contents

1	Introduction	2
2	Command-line options	2
3	Interface	2
4	GUI Console	3
5	Example	4
6	Compilation problems	6

1 Introduction

The *Property Server* can be used as a very simple database utility, the registry or name service.

The *Property Server* allows to define objects (identified by name) and for each object a set of properties (also identified by name) can be created with some values of arbitrary types.

The service stores its data in a text file. The format of this file is not described, but is very easy to add or remove the entries by hand by observing the analogies.

2 Command-line options

The service can be started with the command:

```
$ ./pserver filename ippport [vrq]
```

filename is the name of the file where the data is (or will be) stored. This file can be empty, but it has to exist.

ippport is the port number on which the service's agent will listen for messages.

v means *verbose* – with this option the service will report its actions on the console.

r means *read-only* – with this option the service will not allow the clients to change the data.

q means *quit-enable* – this option causes the server to register the special object that allows the clients to shut the server down.

The **v**, **r** and **q** options are independent from each other.

3 Interface

When run, the *Property Server* creates its own agent and registers an object with the name **propertyserver**.

This object accepts the following messages:

new with one parameter containing the name of new object in the database. If the object with a given name already exists, it will be deleted (with its properties) and new object will be created with no properties. The service replies with the empty *Parameter Set* as a confirmation or rejects if run in the read-only mode.

remove with one parameter containing the name of the object to be removed. It is not an error if the object with a given name does not exist when this message is received. The service replies with the empty *Parameter Set* as a confirmation or rejects if run in the read-only mode.

list (*Parameter Set* is ignored, even if passed) – the server replies with the *Parameter Set* containing all the object names or the empty *Parameter Set* if no name is defined.

write (*Parameter Set* is ignored, even if passed). The service will write the data to the file and reply with the empty *Parameter Set* or reject if run in the read-only mode.

When run in *quit-enable* mode, the server additionally registers an object with the name `propertyserver_quit`. Any message sent to this object will cause the server to shut down.

For each object created in the server (see **new** message above), the service accepts the following messages:

set with two parameters, the first defining the property's name and the second carrying its value. If the property with a given name already exists, it will be replaced. The service replies with the empty *Parameter Set* as a confirmation or rejects if run in the read-only mode.

get with parameters containing the names of requested properties. The server replies with the corresponding values.
 If the query is for one property only and there is no such property, the server replies with an empty *Parameter Set*.
 If the query is for many properties and at least one property cannot be found, the server rejects.

remove with one parameter containing the name of the property to be removed. It is not an error if the property with a given name does not exist when this message is received. The service replies with the empty *Parameter Set* as a confirmation or rejects if run in the read-only mode.

list (*Parameter Set* is ignored, even if passed) - the server replies with the *Parameter Set* containing all the property names or the empty *Parameter Set* if no property for a given object is defined.

4 GUI Console

To facilitate exploring the server's data, the Tcl/Tk GUI console is provided.

After starting, the user is asked to provide the address and port of the *Property Server*. The port should be the same as the one provided as a parameter to the server's main process.

The console displays a list of objects defined in the *Property Server*. After double-clicking one of the objects, the list of properties is filled with all the property names defined for selected object. Objects and properties can be added or removed with appropriate buttons. The property part of the console contains the entry field for the property's value and the radio button for its type. For the moment, the console does not allow to set wide strings and binary values.

Additional buttons are provided to allow the user to write data to the server's file and shut the server down.

5 Example

Let's suppose we want to use *Property Server* as a Name Service.

The server will be used as a common repository of configuration information for the whole distributed system.

Let's suppose we have two servers in the system. The server ABC runs at `comp.mycompany.com` and port 12340. The server XYZ runs at `10.1.0.1`, port 12345 and has level 1. If we want to store all these properties in the *Property Server* for use by other components of the distributed system, we can proceed with these steps:

1. Run the property server with `vq` options. The `v` option will cause the server to print all its actions on the screen. The `q` option will allow us to cleanly shut the server down.
2. Run the GUI console. Provide address and port of the property server. In the console, add new object named ABC and for this object add two properties: `address` with string value `comp.company.com` and `port` with integer value 12340. Add another object XYZ with properties: `address` with string value `10.1.0.1`, `port` with integer value 12345 and `level` with integer value 1. After that, write the data to file and shut the server down.
3. Run the server again, with `r` option. Thanks to this, no client will be able to change any information in the server and the server will run forever, until killed by the administrator.
4. Start up the whole system. Every component, which is aware of the *Property Server*'s location, can send a `get` message to the ABC or XYZ objects on the server with parameters (`address`, `port`). The server will reply with appropriate values. In addition, the XYZ object can be queried for its `level` property. In fact, the query can be for any set of defined properties.

In the scheme described above, the data file used by server contains:

```

ABC
2
address
  s 1
  comp.046mycompany.046com
port
  i 12340
-----
XYZ
```

```

3
address
  s 1
  10.0461.0460.0461
level
  i 1
port
  i 12345
-----

```

Each object entry is ended by a separator line. Below the object's name, there is a number of properties defined. For each property, there is a name, its type and value. For string values, there is a number of lines and the text, all lines are concatenated (only alphanumeric characters are stored explicitly, others are encoded as a dot and three-digit decimal value; the original '.' character is encoded as .046 here, because 46 is the code for '.'; similarly, spaces are encoded as .032 and so on). For binary values, there is a number of bytes and a list of decimal values.

The explanation above is not the formal description, so it is preferred to use property server itself to prepare its files. However, simple changes can be done with any text editor.

As an additional example, here is the code in Python that can feed the server with the necessary information (suppose that the *Property Server* itself runs on local machine and on port 12340; note also that this code does not check for responses from the server, it does not even receive any responses):

```

>>> from YAMI import *
>>> a = Agent()
>>> a.domainRegister('ps', '127.0.0.1', 12340, 2)
>>> a.sendOneWay('ps', 'propertyserver', 'new', ['ABC'])
>>> a.sendOneWay('ps', 'ABC', 'set', \
... ['address', 'comp.mycompany.com'])
>>> a.sendOneWay('ps', 'ABC', 'set', ['port', 12340])
>>> a.sendOneWay('ps', 'propertyserver', 'new', ['XYZ'])
>>> a.sendOneWay('ps', 'XYZ', 'set', ['address', '10.1.0.1'])
>>> a.sendOneWay('ps', 'XYZ', 'set', ['level', 1])
>>> a.sendOneWay('ps', 'XYZ', 'set', ['port', 12345])
>>> a.sendOneWay('ps', 'propertyserver', 'write')
>>> a.sendOneWay('ps', 'propertyserver_quit', 'quit')

```

(the last message shuts the server down)

The code that asks the Name Server for the port of the XYZ component can look like here:

```

>>> from YAMI import *
>>> a = Agent()

```

```
>>> a.domainRegister('ps', '127.0.0.1', 12340, 2)
>>> msg = a.send('ps', 'XYZ', 'get', ['port'])
>>> msg.getResponse()
[12345]
>>>
```

It should be easy to translate this example to other languages.

The example above also shows that YAMI is an excellent tool for remote administration.

6 Compilation problems

g++ 2.95 is known to have poor support for `std::wstring` class, see also notes for *YAMI C++ Wrappers*. On Linux and FreeBSD, the *Property Server* is compiled by default with `-DYAMI_NO_WSTRING` option, so it does not operate correctly on wide strings. If you have access to better compiler, you can remove this option from the respective Makefile.