U.S. DEPARTMENT OF ENERGY

PNNL-22586

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

The Server Name Resolver

DES-0004

Revision 2

Charlie Hubbard

June 2011

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

The Server Name Resolver

Charlie Hubbard

DES-0004
Revision 2
June 2011

DISCLAIMER

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under
Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831-0062
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161
ph: (800) 553-6847
fax: (703) 605-6900
email: orders@ntis.fedworld.gov
online ordering: http://www.ntis.gov/ordering.htm

Contents

1 Introduction

This document describes the Server Name Resolver (*resolverServer.cpp*), its associated client interface library (*resolverClientLib.h*, *resolverClientLib.cpp*), and the resolver test menu client (*resolverMenu.cpp*). The resolver forms an integral part of the inter-process communication (IPC) scheme used in the system control software. Specifically, it provides a mechanism for server applications to register themselves and for client applications to retrieve the information they need to establish a connection to a server.

The primary documentation for this feature is the source code itself and the Doxygen-generated HTML output that is based on the source code. The document you are reading now is supplemental. It exists to provide a deeper background into the design and purpose of the name resolver. If cases are found where the Doxygen documentation disagrees with this document, the Doxygen documentation should be considered correct.

2 History

Originally the system control software was written to run under the QNX operating system. QNX is a *microkernel* architecture operating system, meaning that, internally, the operating system is implemented as a number of small, concurrently running processes, that each perform a specific task (process management, file system management, network driver, etc.), and these processes communicate with one another by passing messages back and forth via the operating system kernel.

This paradigm also proves to be excellent for system control applications, so we adopted it in the form of our *client/server* model architecture. Like the QNX OS, the system control software is comprised of a number of concurrently running applications, each of which is responsible for one piece of the overall system control problem. Individual applications coordinate with each other by passing messages back and forth. Under QNX, this was easy to do. Because QNX relies heavily on message-passing, it provides a robust and easy-to-use set of system library calls specifically for passing messages between processes.

Today however, our system control software is implemented on the Linux operating system. This change from QNX to Linux has partly to do with cost (QNX is expensive, Linux is free), partly to do with customer demand, and partly to do with the rich development environments and toolsets available under Linux that are not available under QNX. Although we gave up QNX, we were loathe to give up the client/server paradigm we use for system control. The client/server model has many advantages over a more traditional monolithic approach, and we wanted to retain those advantages under Linux. Linux provides a number of varied IPC mechanisms, but they all exist on a somewhat more primative level compared to what QNX offered. That is to say, the building blocks necessary to provide QNX-like client/server message-passing are all there, but it is up to the software developer to add features like server discovery, automatic client reconnection, message serialization, and so on.

One feature of the QNX IPC mechanism we appreciated is that it could just as easily send

messages between two processes running on different computers as it could between two processes running on the same computer. This ability had proven handy in the past, so it was an ability we wanted to retain. For that and other reasons, we chose to build our custom IPC mechanism on top of the Internet standard Transmission Control Protocol (TCP).

But that only solves part of the problem. In order for a client process to talk to a server process, it first needs to establish a link to the server process (in the QNX vernacular, these links are called *virtual circuits*). Under QNX, there were built-in facilities in place to make this easy. Specifically, under QNX, all server applications assign themselves a text name. When they start, they register that name with the operating system. Later, when a client wants to communicate with a server, it asks the operating system to establish a virtual circuit connection to the server that registered with that name. Then the operating system does the dirty work to establish the messaging connection between those two processes. Linux has no equivalent built-in mechanism, so we have had to write our own. That mechanism is the Server Name Resolver.

3 Resolver Overview

The Server Name Resolver is a stand-alone network server application that, on start up, attaches itself to a standard TCP port (we typically use TCP port 8686 for this) and then waits for clients that require name resolution services to connect to the port and make requests. There are only a few requests that clients can make, and those are summarized in plain language below.

1. I am a new server just starting up, and I want to register myself as *name*.
2. I am a server in the process of shutting down. Remove my name from your table.
3. I am a client, and I want to communicate with a server called *name*. Give me the appropriate connection information for that server.
4. Return a list of all server names (and associated connection information) currently registered.

As you can see, the resolver application provides roughly the same kind of capability to clients and servers that they had under QNX. On startup, new servers can register themselves, and they can unregister themselves when they shut down. Also, clients interested in communicating with a particular server can get the necessary information to establish a connection to that server from the resolver.

Our approach differs slightly from the QNX approach in that the resolver can only supply a client with the information it needs to establish a connection to a server. It can't actually go ahead and establish that connection the way QNX would. Instead, once the client has the information, it is responsible for establishing the communication channel to the server itself. In practice, this is no great hardship, because the task of establishing and maintaining connections to servers is handled transparently by the `BaseClient` class (a C++ base class

from which all real clients are derived — see design document ***DES-0005, Client/Server Architecture*** for details), so client writers don't have to worry about it.

4 Resolver Server Implementation

4.1 Usage

The resolver server application is implemented in the file *resolverServer.cpp*. The application itself is started from the command line with the following syntax.

```
resolverServer port
```

where `port` is the TCP port number on which the application should listen for client connections.

When the resolver first starts, it uses the standard sockets library to attach itself to the user-specified TCP port (by convention, we use port 8686), and waits for clients to connect. From that point on, clients and servers that need name resolution services can connect to the resolver server and send the appropriate request messages.

4.2 TCP Connection Specifics

Two standard flags are set on the TCP connection — `SO_REUSEADDR` and `TCP_NODELAY`. The first allows the resolver to be stopped and immediately restarted without having to wait the normal TCP 2MSL timeout period before the port can be used again. The second disables the TCP protocol's Nagle algorithm so that it will transmit data as soon as it becomes available rather than delaying some small amount of time in the hopes that more of the available buffer space can be filled before transmitting.

4.3 The Lookup Table

The resolver uses a lookup table to associate a server's text name with the information necessary to establish a connection to the server. The lookup table is implemented using a global STL map. The map is keyed on server text name, and each entry contains a `ServerRecord` struct that contains the connection information. Both the record and the map are defined in *resolverClientLib.h* with the definition reproduced below.

```
struct ServerRecord {
   string serverName;
   string address;
   WORD   port;
```

```
    string Serialize() const;
    void   Deserialize(const string &s);
};
typedef map<string, ServerRecord> ServerMap;
```

4.3.1 The `Serialize()` Method

The `Serialize()` method is a convenience function written to support serializing the contents of a `ServerRecord` into a string using a format compatible with our *Data Serialization Protocol* (see design document ***DES-0002, Data Serialization Protocol*** for details). This is used by the resolver server when building response messages.

4.3.2 The `Deserialize()` Method

The `Deserialize()` method is the inverse of the `Serialize()`. It takes as its one and only input, a string of the format generated by `Serialize()`, and populates the structure fields with the information parsed out of that string. This method is used by the `ResolverClient` class (see section 5, 5) when interpreting responses from the resolver server.

4.4 Threaded Operation

The resolver is capable of handling multiple client connections simultaneously. To do that, each time a client connects, a new communication thread is spawned to handle that client's session, then the main line code goes back to listening for additional client connections. Threading is accomplished using the standard POSIX *pthreads* library. The per-client communication threads are all instances of the `ConnectionHandler()` function.

Because the per-client communication threads all rely on the same global server name lookup table, a mutex is necessary to ensure that only one thread can manipulate the lookup table at a time. The mutex is simply called `mutex` in the code, and, like the lookup table, it is defined globally. All mutex manipulation happens in the `ConnectionHandler()` function, where the mutex is locked before changes are made to the table and unlocked afterward.

4.5 Message Handling

Messages sent to and from the resolver are formatted using our in-house *Data Serialization Protocol* (see design document ***DES-0002, Data Serialization Protocol*** for complete details).

All message handling takes place inside the `ConnectionHandler()` function. As messages arrive on the TCP connection, `ConnectionHandler()` uses an instance of the `ReadStream` class to read them and an instance of the `MessageParser` class to parse them. These two

classes relieve the developer of much of the burden of dealing with the DSP message format. For complete details on these two classes, see design documents ***DES-0002, Data Serialization Protocol*** and ***DES-0003, The ReadStream Utility Class***. As each message arrives, the `ConnectionHandler()` function performs the client's requested action and sends a response back to the client.

The resolver server accepts four messages — *register*, *unregister*, *lookup*, and *serverlist*.

4.5.1 The *register* Message

The *register* message is used by servers that wish to make themselves available to clients. Typically a server will send a *register* message when it first starts up. The DSP formatted string representing the *register* message and its response are shown below.

```
(register,<name>,<port>)
(ok)
```

Here `<name>` is the text name the server wants to register itself as (this field is case insensitive), and `<port>` is the TCP port number that the server will be listening on for client connections. Note that the server does not have to explicitly provide its IP address, because the resolver can glean that from the connection.

On receipt of this message, the resolver will add an entry to its lookup table for a server named `<name>`. Note that if a server by that name is already registered, the old entry in the lookup table will be silently replaced by the new entry.

Note that it is not considered an error to replace an old entry with a new one. In fact, the ability to do so is important because there is no timeout or heartbeat mechanism in place between servers and the name resolver. If a server crashes, it may leave an orphaned entry in the resolver (clients can deal with these properly). If it was not possible for a new server to overwrite old information, there would be no way to restart a server after a crash.

Clearly this can be a source of potential problems. For one, multiple servers with the same name can't co-exist on a system. We deal with this through policy rather than enforce a solution through software (that is to say, the rule is *"No two system control servers can have the same name."*). Also, because there is no authentication to the resolver, malicious code posing as a server could hijack a connection. This is potentially serious, but we assume that we have complete control over all code running on the local control system, and we assume that control systems are either isolated from the general network or connected only to a trusted network, so they can't be tampered with by outside influences.

4.5.2 The *unregister* Message

The *unregister* message is used by servers when they shutdown to inform the name resolver that they will no longer be available to handle client requests. The DSP formatted string

representing the *unregister* message and its possible responses are shown below.

```
(unregister,<name>)
(ok)
(error,'<name>' is an unknown server)
```

Here `<name>` is the text name of the server making the request. On receipt of this message, the resolver removes the entry for the server from its lookup table.

Again, this message is dangerous in that malicious code could use this message to make a server unreachable by clients, but, for the reasons stated above, we discount this as a serious shortcoming. Still, it is something to keep in mind if the assumptions about network isolation and/or security we make above become invalid.

4.5.3 The *lookup* Message

The *lookup* message is used by clients when they want to establish a communication link with an existing server. The DSP formatted string representing the *lookup* message and its possible responses are shown below.

```
(lookup,<name>)
(ok,(<name>,<ip>,<port>))
(error,'<name>' is an unknown server)
```

Here `<name>` is the name of the server being looked up, `<ip>` is the IP address of the system the server is running on (in dotted decimal format), and `<port>` is the TCP port number that the server is listening on for new client connections.

4.5.4 The *serverlist* Message

The *serverlist* message is used primarily by the test menu application to provide a way for developers to see the current state of the resolver's lookup table. Although this message is available to everyone, it is typically used only during development and testing. The DSP formatted string representing the *serverlist* message and its possible responses are shown below.

```
(serverlist)
(ok,(<name>,<ip>,<port>),...,(<name>,<ip>,<port>))
```

Here `<name>` is the name of a particular server registered with the resolver, `<ip>` is the IP address of the system that server is running on (in dotted decimal format), and `<port>` is the TCP port number that the server is listening on for new client connections.

Note that the actual response is variable, containing zero or more individual lookup table records, depending on how many records are currently in the resolver's lookup table.

5 The `ResolverClient` Class

Although it is possible for any process to connect to the resolver server and generate and interpret the resolver's DSP format messages directly, having to deal with the underlying message format is somewhat cumbersome and error prone. Instead, a client API class called `ResolverClient` (defined and implemented in *resolverClientLib.h* and *resolverClientLib.cpp* respectively) has been developed that provides a layer of abstraction between the meaning of the messages and the specific format of the messages themselves. This serves two purposes. First, users of the name resolver can communicate with it just by making simple function calls, with no concern for setting up or tearing down network connections or interpreting arcane message strings. Secondly, by providing this layer of abstraction, we are free to change the underlying message format later on without impacting any existing software (provided the existing software uses the resolver's client interface class).

The definition of the `ResolverClient` class appears below.

```
class ResolverClient {
public:
   ResolverClient(const string host, const WORD port);
   virtual ~ResolverClient();
   bool Connect();
   bool Disconnect();
   bool RegisterServer(const string name, const WORD port);
   bool UnregisterServer(const string name);
   bool LookupServer(const string name, ServerRecord &rec);
   bool GetServerList(ServerMap &sMap);

   string     lastErrorMessage;

private:
   bool ReadResponse(string &resp);

   bool       connected;
   string     hostName;
   WORD       tcpPort;
   int        sockId;
   char       scratch[256];
   ReadStream readStream;
};
```

5.1 The Constructor

```
  ResolverClient(const string host, const WORD port);
```

The constructor takes two parameters. `host` is the name (or dotted decimal IP address) of the computer hosting the resolver service, and `port` is the TCP port the resolver service is listening on for new connections.

The constructor just stores the parameters internally, and marks the connection as being unconnected. It does not actually attempt to establish a connection to the resolver server.

5.2 The Destructor

```
virtual ~ResolverClient();
```

The destructor just checks the current status of the connection to the resolver server and, if it is connected, closes it.

5.3 The `Connect()` Method

```
bool Connect();
```

The purpose of the `Connect()` method is to establish a TCP connection to the resolver server at the IP address and port defined previously (in the constructor).

If the call to `Connect()` is successful, the method returns *true*. Otherwise *false* is returned. If the call fails, the class attribute `lastErrorMessage` will contain a text description of the problem.

5.4 The `Disconnect()` Method

```
bool Disconnect();
```

The `Disconnect()` method checks the current status of the connection to the resolver server and, if it is connected, closes it. This method always returns *true*.

5.5 The `RegisterServer()` Method

```
bool RegisterServer(const string name, const WORD port);
```

The `RegisterServer()` method corresponds to the *register* message defined in section 4.5.1. It is used by new servers to tell the resolver that they are available for client connections. The method takes two parameters. The first, `name`, is the text name the registering server wants to be known by. The second, `port`, is the TCP port the registering server is listening on for new client connections.

If the call to `RegisterServer()` is successful, the method returns *true*. Otherwise *false* is returned. If the call fails, the class attribute `lastErrorMessage` will contain a text description of the problem.

The functioning of the method is simple. First, it checks to see whether there is an active connection to the resolver (an error is returned if there is not). Then it builds a DSP format string for a *register* message, and transmits it to the resolver server. Finally, it parses the DSP format response from the resolver. If the server indicated success, then the method terminates, returning *true*. Otherwise, the error message reported by the server is made available to the caller through the class' `lastErrorMessage` string, and *false* is returned.

5.6 The `UnregisterServer()` Method

```
bool UnregisterServer(const string name);
```

The `UnregisterServer()` method corresponds to the *unregister* message defined in section 4.5.2. It is used by existing servers to tell the resolver that they will no longer be available to service client connections (usually because they are shutting down). The method takes one parameter, called `name`, that should be filled in with the name the server used when it originally registered itself with the resolver.

If the call to `UnregisterServer()` is successful, the method returns *true*. Otherwise *false* is returned. If the call fails, the class attribute `lastErrorMessage` will contain a text description of the problem.

The functioning of the method is simple. First, it checks to see if there is an active connection to the resolver (an error is returned if there is not). Then it builds a DSP format string for an *unregister* message, and transmits it to the resolver server. Finally, it parses the DSP format response message returned by the resolver server. If the server indicated success, then the method terminates, returning *true*. Otherwise, the error message reported by the server is made available to the caller through the class' `lastErrorMessage` string, and *false* is returned.

5.7 The `LookupServer()` Method

```
bool LookupServer(const string name, ServerRecord &rec);
```

The `LookupServer()` method corresponds to the *lookup* message defined in section 4.5.3. It is used by clients to request the information they'll need to establish a connection to an existing server. The method takes as its one input parameter, the text name of the server it wants to communicate with. The method also has one output parameter, `rec`, which will be populated with the requested information if the call completes successfully (returns *true*). If the call fails, *false* is returned and the class attribute `lastErrorMessage` will contain a text description of the problem.

The functioning of the method is simple. First, it checks to see if there is an active connection to the resolver (an error is returned if there is not). Then it builds a DSP format string for a *lookup* message, and transmits it to the resolver server. Finally, it parses the DSP format response message returned by the resolver server. If the server indicated success, then the caller-provided `rec` parameter is populated and the method terminates, returning *true*. Otherwise, the error message reported by the server is made available to the caller through the class' `lastErrorMessage` string, and *false* is returned.

5.8 The `GetServerList()` Method

```
bool GetServerList(ServerMap &sMap);
```

The `GetServerList()` method corresponds to the *serverlist* message defined in section 4.5.4. It is used primarily by the resolver menu test application to check the contents of the resolver's internal lookup table. The method takes one output parameter, called `sMap` that, assuming the call is successful, will be populated with the contents of the resolver lookup table. If the call fails, the class attribute `lastErrorMessage` will contain a text description of the problem, and the value of `sMap` is left undefined.

The functioning of the method is simple. First, it checks to see if there is an active connection to the resolver (an error is returned if there is not). Then it builds a DSP format string for a *lookup* message, and transmits the message to the resolver server. Finally, it parses the DSP format response message returned by the resolver server. If the resolver indicated success, the caller-provided `sMap` field is populated with the data the resolver provided, and the method terminates, returning *true*. Otherwise, the error message reported by the resolver is made available to the caller through the class' `lastErrorMessage` string, and *false* is returned.

6 The `resolverMenu` Program

The `resolverMenu` program (implemented in *resolverMenu.cpp*) is a simple, text-based terminal menu program designed primarily for use during development and testing of the resolver server and its client interface library (described in section 5).

6.1 Usage

The resolver menu application is a command-line utility started with two mandatory parameters as shown below.

```
resolverMenu host port
```

Here `host` is the host name (or IP address in dotted decimal format) of the machine hosting the resolver server, and `port` is the TCP port the resolver server is expecting new client connections to appear on.

6.2 Operation

If the resolver server is not running at the time the menu application is started, the application will terminate with an error (this is different behavior than exhibited by most of the other system test menu programs, which start up okay, even if the server to which they communicate is not running). Once started, the application displays a simple text mode menu and a prompt as shown below.

```
 1 - Register a server
 2 - Unregister a server
 3 - Lookup server
 4 - Get server list

Enter Selection >
```

As example above demonstrates, the application provides one option for each message the resolver server can respond to. Selecting a particular option will cause the user to be prompted for any additional information necessary for the corresponding message type, and then the message will be sent to the resolver and the resolver's response will be displayed back to the terminal. We will not cover every option in this document, primarily because the operation of the utility is pretty self-explanatory. However, we will close the section with the screen output as it might appear after requesting the resolver's lookup table on an active system.

```
 1 - Register a server
 2 - Unregister a server
 3 - Lookup server
 4 - Get server list

Enter Selection > 4
Server 'anadig' is at 127.0.0.1 on port 1458
Server 'control' is at 127.0.0.1 on port 3034
Server 'logger' is at 127.0.0.1 on port 5462
Server 'mks' is at 127.0.0.1 on port 2229
Server 'pid' is at 127.0.0.1 on port 4092
Server 'soh' is at 127.0.0.1 on port 5060
Press any key to continue...
```

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by **Battelle** *Since 1965*

U.S. DEPARTMENT OF
ENERGY

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99352
1-888-375-PNNL (7665)
www.pnnl.gov