Neptune Logging

1 Overview

Neptune includes a logging framework that allows client code to log certain events. Application code uses the logging macros to log events through logger objects.

Logger objects have a unique name and are responsible for dispatching log events through the logging framework.

A log event has a log level that indicates a notion of priority or severity, a source filename, a source line number, and a message.

Loggers encapsulate event information in an event record, which contains all the parameters supplied by the client, as well as a timestamp and the name of the logger through which the event was initially dispatched.

Loggers are organized in a hierarchical tree structure. Each logger has its own configuration that includes a log level used to filter log events (a logger will only dispatch log events which have a log level equal or higher to the logger's log level), one or more handlers, and a parent logger. Loggers will, by default, forward all of their log event records to their parent logger, unless their configuration specifies not to forward records. When a logger is not configured with its own log level, it inherits the log level of its parent.

The parent-child relationship of loggers is determined by the name of the loggers. A logger name is a list of one or more name fragments separated by a dot. The parent of a logger is the logger with the name equal to that logger's name excluding the last name fragment (for example, the parent of the logger named 'neptune.test.foo' is named 'neptune.test'). The framework supplies a root logger as an ancestor of all other loggers in the system. The root logger's name is an empty string.

Log handlers are responsible for formatting log event records and publishing them. The framework includes built-in handlers, including a handler that formats log event records as text and writes them to a file or to the console, and a handler that formats log event records as a message and sends it to a logging console over a network connection.

Logger objects are automatically created on behalf on the client code the first time they are used. They are automatically destroyed when the application exits.

The configuration of all the loggers in the system, including their log levels, list of handlers, and handler options (such as the name of the file to write to for file-based log handlers) is encoded in a logging configuration file and/or in a property list. The configuration source for the system can be specified at compile time, or at runtime through the use of environment variables.

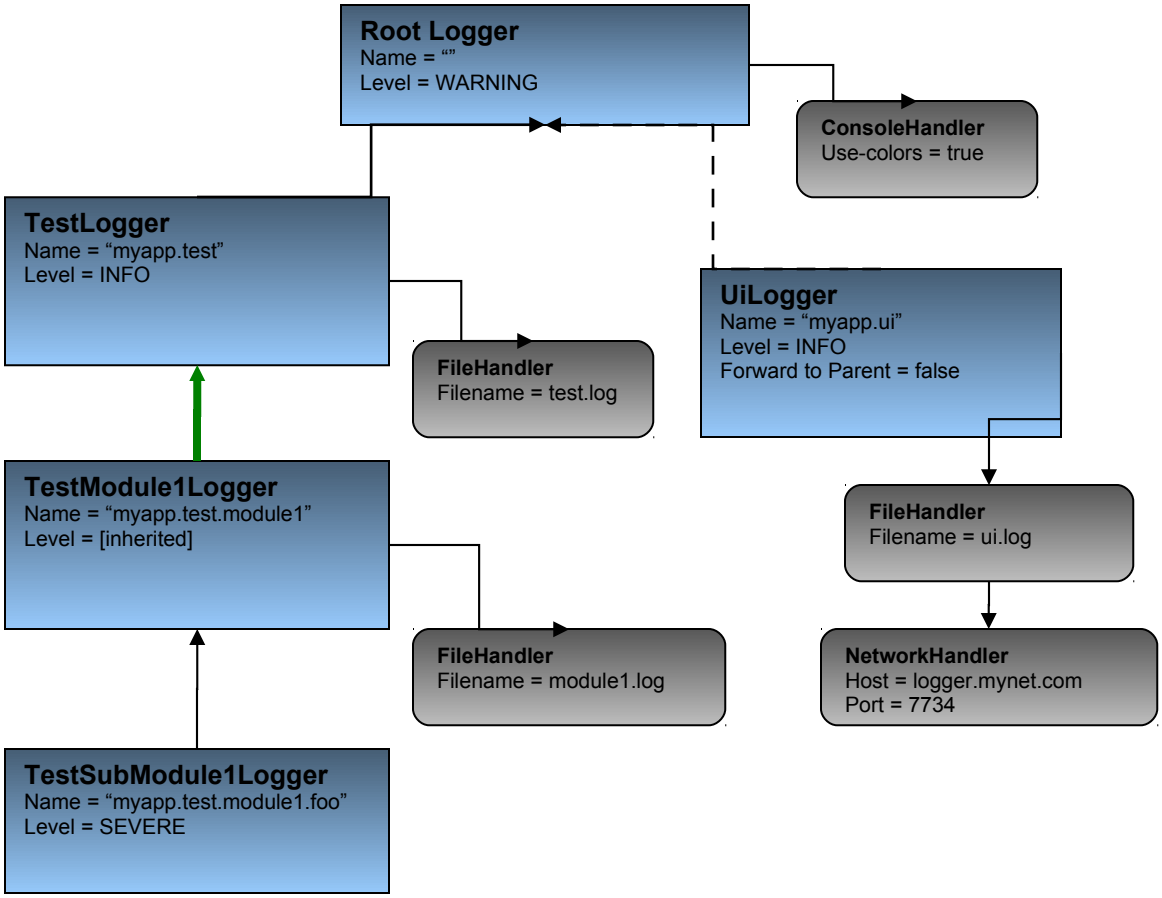

Figure 1 – Logging configuration example

In this example, the root logger is configured to log all events with level WARNING or above to the console, using console colors. The TestLogger logger will log all log events with level INFO or above to the file 'test.log' and then forward them to the root logger. The TestModule1Logger inherits its log level from its parent, and writes all formatted log records to the file 'module1.log' before forwarding them to its parent. The logger TestSubModule1Logger does not have its own handler, but has its own log level which will filter out all log events with a level lower than SEVERE. Finally, the UiLogger logger will log all events with level INFO or above to the file 'ui.log' as well as to the log console running on host 'logger.mynet.com' on port 7734. It will not forward any event to its parent.

2 Client API

The client code can log events by using any of the logging macros. To log an event, the client code needs to specify a logger to use. The logging macros come in two flavors: the local-logger macros, which log to the local logger defined for this file and the specific-logger macros, which take as their first argument the name of a logger variable that identifies a specific logger. Both styles of macros either have an implicit log level, or an explicit log level. Finally, there are 10 versions of each macro, one for each size of the argument list used to construct the log message, from 0 arguments to 9 arguments. When the macro has message arguments, the message string is a printf-style format where the % placeholders will be replaced by the formatted argument values.

2.1 Log Levels

Log levels are positive integer numbers; the higher the number is, the higher the priority is. Symbolic names have been assigned to predefined log levels for convenience. The predefined log levels for log events are:

```
#define NPT_LOG_LEVEL_FATAL   700
#define NPT_LOG_LEVEL_SEVERE  600
#define NPT_LOG_LEVEL_WARNING 500
#define NPT_LOG_LEVEL_INFO    400
#define NPT_LOG_LEVEL_FINE    300
#define NPT_LOG_LEVEL_FINER   200
#define NPT_LOG_LEVEL_FINEST  100
```

It is possible, but not recommended, to use log levels other than the predefined ones.

In addition, the log level of loggers can be set to accept all, or reject all, log events, using the following constants:

```
#define NPT_LOG_LEVEL_OFF     32767
#define NPT_LOG_LEVEL_ALL     0
```

2.2 Local Logger Macros

There can only be one name per C/C++ source file for the local logger. To define the local logger for a source file, declare the local logger name at the top of the file, with the following macro:

```
NPT_SET_LOCAL_LOGGER(name)
```

Example:

```
NPT_SET_LOCAL_LOGGER("myapp.test")
```

NOTE: do not put a semicolon (';') character at the end of the macro, because the macro would expand to a non-ansi-compliant syntax that some compilers will complain about.

The local logger macros are:

```
NPT_LOG(level, msg)
NPT_LOG_1(level, msg, arg1)
NPT_LOG_2(level, msg, arg1, arg2)
NPT_LOG_3(level, msg, arg1, arg2, arg3)
NPT_LOG_4(level, msg, arg1, arg2, arg3, arg4)
NPT_LOG_5(level, msg, arg1, arg2, arg3, arg4, arg5)
NPT_LOG_6(level, msg, arg1, arg2, arg3, arg4, arg5, arg6)
NPT_LOG_7(level, msg, arg1, arg2, arg3, arg4, arg5, arg6, arg7)
NPT_LOG_8(level, msg, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)
NPT_LOG_9(level, msg, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9)
```

In addition, for each predefined log level, there are 10 macros with implicit level, one for each number of arguments. Those macros are:

```
NPT_LOG_FATAL(msg), NPT_LOG_FATAL_1(msg, arg1),...
NPT_LOG_SEVERE(msg), NPT_LOG_SEVERE_1(msg, arg1),...
NPT_LOG_WARNING(msg), NPT_LOG_WARNING_1(msg, arg1), ...
NPT_LOG_INFO(msg), NPT_LOG_INFO_1(msg, arg1), ...
NPT_LOG_FINE(msg), NPT_LOG_FINE_1(msg, arg1), ...
NPT_LOG_FINER(msg), NPT_LOG_FINER_1(msg, arg1), ...
NPT_LOG_FINEST(msg), NPT_LOG_FINEST_1(msg, arg1), ...
```

Example:

```
NPT_LOG_1(NPT_LOG_LEVEL_INFO, "the temperature is %d", temperature);
NPT_LOG_WARNING("pay attention!");
```

2.3 Explicit Logger Macros

There can be one ore more explicit loggers for each C/C++ source files. To use an explicit logger, the client code must define one or more logger references. A logger reference defines a logger variable, and a logger name bound to that variable. The logger variable will be automatically initialized to point to the appropriate logger upon first use of a logging macro for that logger.

To define a logger reference, use the following macro:

```
NPT_DEFINE_LOGGER(variable_name, logger_name)
```

Example:

```
NPT_DEFINE_LOGGER(MyTestLogger, "myapp.test")
```

NOTE: do not put a semicolon (';') character at the end of the macro, because the macro would expand to a non-ansi-compliant syntax that some compilers will complain about.

```
NPT_LOG_L(logger, level, msg)
NPT_LOG_L1(logger, level, msg, arg1)
NPT_LOG_L2(logger, level, msg, arg1, arg2)
NPT_LOG_L3(logger, level, msg, arg1, arg2, arg3)
NPT_LOG_L4(logger, level, msg, arg1, arg2, arg3, arg4)
NPT_LOG_L5(logger, level, msg, arg1, arg2, arg3, arg4, arg5)
NPT_LOG_L6(logger, level, msg, arg1, arg2, arg3, arg4, arg5, arg6)
NPT_LOG_L7(logger, level, msg, arg1, arg2, arg3, arg4, arg5, arg6, arg7)
NPT_LOG_L8(logger, level, msg, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)
NPT_LOG_L9(logger, level, msg, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9)

In addition, for each predefined log level, there are 10 macros with implicit level, one
for each number of arguments. Those macros are:

NPT_LOG_FATAL(logger, msg), NPT_LOG_FATAL_1(logger, msg, arg1),...
NPT_LOG_SEVERE(logger, msg), NPT_LOG_SEVERE_1(logger, msg, arg1),...
NPT_LOG_WARNING(logger, msg), NPT_LOG_WARNING_1(logger, msg, arg1), ...
NPT_LOG_INFO(logger, msg), NPT_LOG_INFO_1(logger, msg, arg1), ...
NPT_LOG_FINE(logger, msg), NPT_LOG_FINE_1(logger, msg, arg1), ...
NPT_LOG_FINER(logger, msg), NPT_LOG_FINER_1(logger, msg, arg1), ...
NPT_LOG_FINEST(logger, msg), NPT_LOG_FINEST_1(logger, msg, arg1), ...

Example:
NPT_LOG_L1(MyTestLogger, NPT_LOG_LEVEL_INFO, "the temperature is %d", temperature);
NPT_LOG_WARNING_L(MyTestLogger, "pay attention!");
```

Here is an example of a C source file that uses both local and specific logger macros, in different styles:

```
/*----------------------------------------------------------------------
|   includes
+----------------------------------------------------------------------*/
#include "Neptune.h"

NPT_DEFINE_LOGGER(MyLogger, "neptune.test.my")
NPT_DEFINE_LOGGER(FooLogger, "neptune.test.foo")
NPT_SET_LOCAL_LOGGER("neptune.test")

/*----------------------------------------------------------------------
|   main
+----------------------------------------------------------------------*/
int
main(int argc, char** argv)
{
    NPT_LOG_L(MyLogger, NPT_LOG_LEVEL_WARNING, "blabla");
    NPT_LOG_L2(MyLogger, NPT_LOG_LEVEL_WARNING, "blabla %d %d", 8, 9);

    NPT_LOG(NPT_LOG_LEVEL_WARNING, "blibli");
    NPT_LOG_2(NPT_LOG_LEVEL_INFO, "fofo %d %d", 5, 7);

    NPT_LOG_SEVERE("this is severe!");
    NPT_LOG_SEVERE_1("this is severe (%d)", 9);
```

```
    NPT_LOG_SEVERE_L(MyLogger, "this is severe!");
    NPT_LOG_SEVERE_L1(MyLogger, "this is severe (%d)", 9);

    NPT_LOG_SEVERE_L(FooLogger, "this is severe!");
    NPT_LOG_SEVERE_L1(FooLogger, "this is severe (%d)", 9);

    NPT_LOG_SEVERE("severe");
    NPT_LOG_WARNING("warning");
    NPT_LOG_INFO("info");
    NPT_LOG_FINE("fine");
    NPT_LOG_FINER("finer");
    NPT_LOG_FINEST("finest");

    NPT_LOG_SEVERE_L(FooLogger, "severe");
    NPT_LOG_WARNING_L(FooLogger, "warning");
    NPT_LOG_INFO_L(FooLogger, "info");
    NPT_LOG_FINE_L(FooLogger, "fine");
    NPT_LOG_FINER_L(FooLogger, "finer");
    NPT_LOG_FINEST_L(FooLogger, "finest");

    return 0;
}
```

3 Builtin Handlers

3.1 Null Handler

The Null handler discards all log records. This handler may be used when a compiled application where logging is enabled needs to be run without any logging output.

3.2 Console Handler

The Console Handler formats log records as text and writes them to the application's standard output. On platforms that have terminals or console screens with support for ANSI color escape sequences, the Console Handler can be configured to use different colors for each log level.

The name of the Console Handler is `ConsoleHandler`.

3.3 File Handler

The File Handler uses the same formatting as the console handler but writes its output to a file. File handlers can be configured with a filename to write to. If a filename is not specified, the records will be written to the file <name>.log where <name> is the name of the logger.

The name of the File Handler is `FileHandler`.

3.4 TCP Network Handler

The TCP Network Handler formats log records as an HTTP-like message (each field of the record is encoded as a header line, and the message text is the message body) and sent over a TCP/IP network connection to a log console. The handler configuration can specify the hostname and port for the log console. If not hostname is specified, the connection is made to 'localhost'. If no port number is specified, port 7723 is used. The connection to the handler is done automatically upon dispatching the first log record. If the connection is interrupted, the handler will automatically reconnect.

The name of the TCP Network Handler is `TcpHandler`.

3.5 UDP Network Handler

The UDP Network Handler formats log records in the same format as the TCP Network Handler, but the log records are sent as UDP datagrams. The handler configuration can specify the

hostname and port for the log console. If not hostname is specified, the connection is made to 'localhost'. If no port number is specified, port 7724 is used.

The name of the UDP Network Handler is `UdpHandler`.

4 Configuration

All loggers and log handlers can be configured using a combination of configuration files and property list strings. The logging subsystem will first try to read its configuration from the configuration sources specified in the environment variable `NEPTUNE_LOG_CONFIG` (the name of this environment variable can be changed by defining the macro `NPT_LOG_CONFIG_ENV` to a different name).

If this environment is set, it must contain one ore more configuration source names. If there are more than one configuration source, the configuration source names must be separated by a vertical bar character ('|').

A configuration source name is either:

`file:<filename>`

or

`plist:<property-list>`

A configuration file is a text file that contains one property per line.

A property list is one or more properties. If there are more than one property, they are separated by a semicolon character (';').

Each property is specified as <key>=<value> where <key> is the name of a property and <value> the value for that property.

If the environment variable is not set, the logging subsystem then looks for a file named `neptune-logging.properties` in the current working directory of the running process. This file contains configuration properties as explained above.

Example:

```
file:logging.properties|
plist:myapp.test.level=FINE;myapp.test.handlers=ConsoleHandler,NetworkH
andler
```

4.1 Logger Configuration

A logger <logger> can be configured by specifying one or more logger configuration properties. Each property has a name that starts with <logger>. The logger configuration properties include:

<name>.level

Property name	Property value	Description
<logger>.level	A symbolic or numeric log level. The symbolic values are: FATAL, SEVERE, WARNING, INFO, FINE, FINER, FINEST, ALL and OFF	The log level for the logger. The logger will only handle log events with a level equal or higher to this level. The level can be set to OFF to ignore all log events, or to ALL to handle all log events. When no level property is specified for a logger, the logger inherits its parent's log

		level.
<logger>.handlers	A comma-separated list of handlers. The name of the builtin handlers are: NullHandler, ConsoleHandler, FileHandler, TcpHandler, UdpHandler	The list of handlers to which the logger will publish log event records.
<logger>.forward	A boolean value.	When set to false, the logger does not forward the log event records to its parent. When set to true, the logger forwards log event records to its parent. If this property is not specified, the default behavior of loggers is to forward log even records to their parent.

Boolean values are true or false. False can be written as 'false', 'off', 'no', or '0'. True can be written as 'true', 'on', 'yes', or '1'.

4.2 Handler Configuration

When a logger <logger> is configured with a list of handlers, each handler <handler> can be configured by setting properties with the prefix <logger>.<handler>

4.2.1 Console Handler

Property Name	Property Value	Description
<logger>.ConsoleHandler.colors	A Boolean value	Specifies whether to use ansi terminal color escape sequences.

4.2.2 File Handler

Property Name	Property Value	Description
<logger>.FileHandler.filename	String	Name of the file to write to. If this property is not specified, FileHandler handlers write to a file with the name <logger>.log

4.2.3 TCP Network Handler

Property Name	Property Value	Description
<logger>.TcpHandler.hostname	String	Name of the host on which the log console application is running. If this property is not specified, the default is 'localhost'.
<logger>.TcpHandler.port	Integer	Port number for the TCP/IP connection to the console. If this property is not set, the default is port 7723.

4.2.4 UDP Network Handler

Property Name	Property Value	Description
<logger>.UdpHandler.hostname	String	Name of the host on which the log console application is running. If this property is not specified, the default is 'localhost'.
<logger>.UdpHandler.port	Integer	Port number to which the log record datagrams are send. If this property is not set, the default is port 7724.

Example:

```
### configure the root logger
.level = WARNING
.handlers = ConsoleHandler
.ConsoleHandler.colors = on
### configure the myapp.test logger
myapp.test.level = INFO
myapp.test.handlers = ConsoleHandler,FileHandler
myapp.test.FileLogger.filename = neptune.test.log

### configure the myapp.ui logger
myapp.ui.level = FINE
myapp.ui.forward = false
myapp.ui.handlers = TcpHandler
myapp.ui.NetworkHandler.hostname = logconsole.mynet.com
myapp.ui.NetworkHandler.port = 8000
```