Lustre Trap Collector HLD

Johan Dahlin

February 11, 2008

1 Introduction

The collector is a daemon usally running on a MDS (meta data server) in a Lustre filesystem. The collector and the mointor tool is designed to solve the problem of delivering traps, an error, warning or information from a lustre node to the administrator. The collector is responsible for reading the traps through from a lustre client and to make the available to monitor tool clients, through a HTTP/XMLRPC interface. Traps on the collector node is sent over the network, through rpc calls. When a trap is generated we can see from what node and filesystem it is generated from. This information is sent to the client which provides a view of the traps triggered by lustre nodes.

2 Overview

2.1 System dependencies

- A Lustre enabled kernel with traps support
- Python 2.2
- PyGTK 2.2
- Twisted 1.3.0

2.2 Use cases

2.2.1 Scenario 1 - Trap level generation

This is a normal scenario provides a high level overview when a trap is sent from lustre.

Who	#	Action	Comments
User	1	run lustre-collector	Program executed
Lustre	2	Collector is started up	See Scenario 2 for details
Lustre	3	A trap is generated	
Collector	4	select call wakes gathering class	
Collector	5	trap is read and stored in the program	
Monitor tool	6	A client connects and gets a list of traps	
Collector	7	Trap is sent to the monitor tool	

2.2.2 Scenario 2 - Startup

Startup of the monitor tool, configurations and initial bootstrap.

Who	#	Action	Comments
User	1	run lustre-collector	Program executed
Collector	2	Reading command line options and configuration files	
Collector	3	Start HTTP/XMLRPC server factory	Collector now accept connections
Collector	4	Start trap gathering class	Traps can now be collected
Collector	5	/proc is examined, the trap file is monitored	

2.2.3 Scenario 3 - Router goes down and fires far too many traps

When a network fails, lustre will report all nodes as being disconnected at approximately the same time. If the number of nodes are high, 10 000+ we will generate far too many traps. MAXTRAPS is a variable read from the configuration class.

Who	#	Action	Comments
User	1	run lustre-collector	Program executed
Collector	2	Collector starts-up	See scenario 2 for details
External	3	Router fails	
Lustre	4	MDS node notes that nodes are down	
Lustre	5	Traps are generated and sent	
Collector	6	select call wakes us gathering class	
Collector	7	Gathering class reads all the traps	
Collector	8	Too many traps are read, more than MAXTRAPS	Only max traps are read and stored in the collector
Monitor tool	9	Client connects and gets read all traps	Only MAXTRAPS are sent

2.2.4 Scenario 4 - Client connects and read traps

A detailed view which displays what happens when a client connects and read the available traps.

Who	#	Action	Comments
User	1	run lustre-collector	Program executed
Collector	2	Collector stats-up	See scenario 2 for details
Monitor tool	3	Client connects	
Collector	4	Connection is validated and security checks are performed	
Monitor tool	5	the xmlrpc method getTraps is	Traps for the last 2 hours are requested
Collector	6	Traps for the requested interval are sent to the client	

2.2.5 Installation

In the first phase we're only supporting Linux. Most customers will install the collector by installing an rpm (and it's dependencies) and run. On a Fedora Core 2/RHL3 system it should only be two rpms; One for Twisted and one for the the collector. Install from a tarball for other systems will also be an option.

Who	#	Action	Comments
Administrator	1	Fetch installation program	RPM
Administrator	2	rpm -i	
Administrator	3	vim /etc/lustre-collector/config	Modify the system-wide configuration (can also be done on the command line)
Administrator	4	/etc/init.d/lustre-collector start	Executes lustre-collector
Program	5	Normal operation	

2.2.6 Configuration

There are two ways of configuring lustre-collector. One is modifying the system wide configuration file, usually in /etc or the one in the users home directory. The second way is specifying the options on the command line. All options can be specified on both the command line or in the configuration files.

See section 3.3.2 for a list of configuration options.

2.3 Performance requirements

In most cases the collector daemon will run on a MDS node, which is already very busy. As little IO and CPU as possible should be used. On some systems we will run mirrored on two MDS nodes, so to be able to have a completely redundant monitor framework the client needs to connect to two MDS nodes at once. The number of file systems may vary, in some cases only one file system will be used but in extreme cases we have 100 different file systems.

The performance goals need to meet the following scenarios:

- 100'000 OST nodes
- 2 MDS nodes

In case of critical uses, for example when network equipment goes down or starts to malfunction; the daemon is going to receive many thousands of traps per second. This needs to be handled in a way that does not limit the performance of the machine. It should also filter out the traps and deliver only a certain amount of traps to the clients. The maximum number of traps sent to the client is normally specified in a configuration file or on the command line.

3 Functional specification

3.1 XMLRPC class

3.1.1 Introduction

Base class, inherited from a Twisted class. A so resource object which is attached to a node (eg /) on a http server object. When the url for the resource is accessed, the render method will be with a request object as an argument. The request is a wrapper around the http client connection and contains information such as IP number, http method, POST data etc. All methods prefixed with xmlrpc_ will be available as methods the client can call.

3.1.2 Traps

Traps are tuples (immutable sequences in python) with the following data included:

- message -> string
- integer -> category, one of the following; 0 -> Info, 1 -> Warning, 2 -> Critical, 3 -> Fatal
- nodeName -> string

Method descriptions

function name	**arguments**	**return type**	**description**
getNodesNames*	fsName -> string	list of strings	Returns a list of node names, if fsName is not empty it will filter the nodes by their respective file system. If no nodes are found, an empty list will be returned
getNodeInfo*	nodeName -> string	dictionary	Returns a dictionary containing node information such as free disk space. If a node with nodeName does not exist, an exception will be raised
getFileSystems*	none	list of strings	Returns a list of file system names or an empty string if no file systems are found
getTraps*	time -> integer	list of 3-sized tuples and time stamp	Returns a list of traps and a time stamp. If no traps are found since time an empty list is returned. The time stamp marks when the trap list was read.
getFileSystemInfo*	fsName ->string	dictionary	Returns a dictionary containing the total number of traps and names of the nodes which are not responding.
render	Twisted Request object	string	This is by the Twisted framework, it's the entry point for a HTTP Request and overridden to add extra security checks.

Note: * xmlrpc_ prefix omitted.

3.2 Trap gathering class

This is reading /proc through a library/interface and stores parts of it in lists which the client can access. This class is an intermediate storage for traps gathered and not filtered. It's used by the XMLRPC/Web class which calls getTraps. When instantiated the program is scheduled to call every n seconds, as specified in the configuration.

function name	**arguments**	**return type**	**description**
	none	none	read traps from /proc and store them in this class.
readNodeInfo	nodeName -> string	dictionary	Returns a dictionary containing node information such as free disk space. If a node with nodeName does not exist, an exception will be raised
getTraps	time -> integer	list of traps	Reads all traps that was triggered since time and return them on a list.

3.3 Config/command line parsing

3.3.1 Class description

Handle configuration file and command line parsing. Instantiated from the main class and only used during startup.

function name	**arguments**	**return type**	**description**
addOption	optionName -> string optionShortName -> char optionType -> type optionDescription -> string	nothing	adds a configuration option
parseOptions	argument list -> variable	nothing	parses configuration options specified on the command line
parseFile	filename -> string	nothing	parses filename and updates internal values
getOption	optionName -> string	value of option with name optionName	

This will be used to load defaults from the configuration file and options specified on the command line.

3.3.2 Configuration options

Configuration is stored in a system wide location and can be overridden by user by creating a file in the home directory (or similar on non-unix systems). Configuration format is suggested to be ini files, since that is the easiest one to implement. XML is another possible way, but not strictly needed in this case, since the amount of configuration is rather limited. Options can also be specified on the command line, in that case the will take precedence over both the system wide and the local configuration file.

Name	**Short name**	**Type**	**Default**	**Description**
port	p	integer	8980*	port to bind
host	h	string	localhost	hostname to bind to can be empty to listen to all available interfaces
daemon	d	bool	false	daemonize
filename	f	string		configuration file to load, overrides configuration file in users home directory
maxtraps	m	int	250*	maximum number of traps that are stored in the collector

* Randomly chosen.

3.4 Main class

The main class is responsible for calling the other classes; starting the web server, parsing configuration data

function name	**arguments**	**return type**	**description**
parseConfiguration	none	none	parses configuration data
startWebServer	none	none	starts the Twisted web server
startCollector	none	none	starts the lustre collector interface
run	none	none	

3.5 Portable Proc Library

Holds the other instances, instantiates them and allows apache to connect etc security contexts

We will be using a library to access the contents of /proc, to be able to be portable, once lustre is ported to MacOSX and win32 it'll be needed.

int llmt_read_file(const char *pathname, int *size, char **data);

int llmt_list_directory(const char *pathname, struct dir_st **directory);

4 Logic Specification

4.1 Data locking & Threads

We're already using Twisted to do asynchronous IO over the network, so there is strictly no need to use threads to collect data from /proc. Since all of the data is cached on the node, we don't need to do any rpc calls to fetch them from the remote nodes. Scanning /proc will be relatively fast and while it's likely to add some latency, clever programming can help us avoid using threads and simplify the design of the collector. Twisted is already running a select loop which we can just append more file descriptors and get notification from /proc when new traps occur.

4.2 Fail over

The collectors are not aware of each other, so they're not trying to synchronize state. To avoid showing traps twice, we can identify them by an id, that should probably be done in kernel space if so.

4.3 Reconnecting

Re-connecting is done on client side, when it dies, it reconnect and fetches the data again. Eventual traps which happened between that point will be resent automatically. (since the client asks for traps since a certain point in time)

4.4 Connecting / Cleanup

This is completely abstracted away by the HTTP server class in Twisted. From the collectors point of view there's no state for any of the connections.

4.5 Connection handshake

Using a normal HTTP connections, client send a request with headers and in the payload section it'll include the xmlrpc method call to getTraps. The server will respond normally with headers and the return value from getTraps.

4.6 Security handling

Once the request object has been given to us we check if it's origin IP address is not 127.0.0.1 or if the request type is not POST we're denying the request. We override the the request for the top level object in the web server instance and add two simple checks (ensure that it comes from the allowed IP/IP range and that it's POST request) and then give the request back to the XMLRPC resource class within xmlrpc. In there it processes the method call and if it's a method that exists it calls xmlrpc_ in the subclass. Otherwise it just raises a failure. Finally before calling xmlrpc it checks for the number of arguments of the server implementation of the rpc, if it's not enough, too many it sends back an error code.

Traps will be delivered in such a way that you can find out which node and file system they originated from. The collector daemon will never be run as root, so using resources that are available only to the super user is out of the question. Clients will never connect directly to the collector they will instead authenticate and connect by a proxy module in apache. The collector should be configured to only allow local connections or connections from the host apache is installed on.

5 State Management

Only state we process are the proc data structures from lustre. The collector daemon does not save state to disk.

5.1 Connections

Due to how xmlrpc is implemented on the server side in Twisted connections are, from the collectors point of view stateless. However they still need to regularly poll the server, by doing some method calls to see if anything happened, something that's unavoidable with xmlrpc. Calls are only client->server. Connections are HTTP 1.1 persistent connections, so when the call is done by the client the connection is not closed and the client can make a new call when he desires. It will avoid the extra overhead of authenticating and establishing a connection for each request, but it will not save us from polling.

6 Limitations of XMLRPC and potential alternatives

The protocol we use to do RPC method calls; XMLRPC has some limitations which might be troublesome; specifically:

XMLRPC does not allow complex objects (such as instances) to be sent over the wire. XMLRPC uses normal HTTP requests, so to be able to send information from the server to the client we need to poll in the client. This is an issue if we want to show the information (eg traps) *immediately.* However in many cases a delay of 5-10 seconds is not a big issue

- **CORBA** - A protocol like CORBA could be used, but CORBA is very big (however easy in Python) but it's non-asynchronous nature forces both the server to use threads to be efficient and responsive, which makes the application more complicated than it needs to be.
- **Perspective Broker** - The network framework we're using for the collector Twisted provides an asynchronous RPC part Perspective Broker. Which can do two-ways communication, send complex objects and is designed with security in mind. However, indifferent to XMLRPC it's not an extension of HTTP so we cannot reuse the apache authentication mechanisms if we choose to use Perspective Broker.
- **Custom protocol** - Writing a custom protocol; or an XMLRPC implementation which uses persistent connections is another options if the latency is to be considered a serious problem

7 Open Issues / Focus for inspection

- Portable Proc Library in C or Python.

- XML and ini as markup configuration storage.
- Move time stamp to a getLastTimestamp() xmlrpc method.
- Port number 8980.
- Default maximal number of traps stored.

Brief key concerns to the attention of the reviewers.