LogService manual

Abstract

LogService is a tool to monitor distributed applications. It provides services to collect messages from different components of the application, to filter this data and to offer it to interested clients. It thus acts as a common base between the application and it's monitoring programs, simplifying the development for both sides.

This manual explains the concepts of LogService, the provided APIs and the installation and usage of the monitor. It further documents the supporting libraries which simplify the development of tools and instrumentation code.

Contents

1   Overview

LogService is a tool that can monitor distributed applications. It acts as a common base with well defined interfaces between the application and it's components on the one side and the tools who monitor the application on the other side. LogService itself consists of two parts, the main program LogCentral (often referred as monitor) that implements the monitoring functionality and two libraries. These libraries are the LogToolBase and the LogComponentBase. They can be used to simplify the development of tools and the instrumentation of application code by providing APIs to communicate with the LogCentral in C++ or Java.

Note that the usage of these libraries is optional, the connection to the core can also be implemented directly by the application using the provided idl interfaces. The features offered by the LogCentral include:

- Communications in CORBA
  Both tools and application communicate with the LogCentral using CORBA. This is a very flexible approach since CORBA does not depend on certain platforms. Using the provided idls, almost any kind of application or tool can connect to LogCentral.

- Classification of messages
  Each information processed by the core is exchanged as a 'message'. Besides their information, these messages provide two fields 'component' and 'tag'. These allow the introduction of a general filter system to classify the messages in two dimensions.

- Filtering of messages
  Based on the possibility to classify messages, each connected tool can define it's own set of filters to define it's own view on the application. The monitor guarantees that each tool gets exactly the messages it wants.

- Prefiltering of messages
  One problem when instrumenting an application is that a part of the information gathered is only needed by very few tools which are not always connected. As sending messages is costly, the core offers methods to allow a prefiltering of messages in the application, minimizing the number of messages that have to be sent.

- Message ordering
  Ordering of messages is always complicated in distributed environments. LogService uses timestamps in conjunction with internal synchronisation mechanisms to allow proper message ordering. It also offers the possibility to hold back messages for a short period of time to compensate minor lags and deliver a sorted stream of messages to the tools.

- Hold system state
  To allow tools to connect at any time, LogCentral holds the state of the monitored system. This does not only include which components are currently connected, it can also contain information on details of the system which are not directly known to LogCentral.

- Proxy forwarder
  To monitor each diet component, even when they are on machines hidden behind firewall. A forwarder must be launched between the two sides to be usefull, multiple forwarders may be used to jump between various firewall.

2   Basic concepts

2.1   The filter system

As already mentioned in the overview, messages are classified by two properties, 'tag' and 'component'. This section explains what these properties are and how they can be used to filter messages on a very general base.

2.1.1   Components

Let's first look at the components. No distributed application consists of only one single program on one host, but of several parts running in different processes on different hosts. When monitoring the

application one is usually interested in where exactly an event occured. The problem is that the different programs of the application can further be divided into distributed objects, logical objects, implemented objects and functions. The needed granularity of monitoring highly depends on the actual instrumentation of the application code and must not be restricted by the monitoring system.

LogService solves this problem by just offering logical components. The user decides to which parts of the application each component corresponds. By doing so he defines a view on the application. Although it is not necessary, it is recommended that the application is monitored with constant granularity. Monitoring some parts of the application with higher granularity than other parts works, but may lead to confusion when building tools. It is also recommended to leave the granularity on a relatively coarse level as it can be refined further by the 'tag' field.

In practice, each component is identified by it's name. This name can either be set by the programmer or generated automatically by the LogCentral. Each component connects and disconnects independently to the LogCentral and can send it's own messages and receive it's own set of prefilters.

2.1.2 Tags

Each component usually generates a number of different messages. These messages can be distinguished by the 'tag' field. It defines what kind of information this messages carries. For example a component that represents a server may generate two different tags to indicate the start and the end of a client request. Just like the components define a view based on the origin of a message, the tags define another view based on the type of the message. This type corresponds to the function of their origin. It can be used to monitor certain functionality of the application without explicitly choosing each necessary component. If we want to monitor all servants of the example above, we just filter for their tags and to get all relevant messages.

Although tags and components are not completely independant, they can be used together to define a very granular and flexible way of filtering. The component denotes the coarse origin of a message and the tag further refines this information.

2.1.3 Messages

Based on this information we now define the messages. Each message contains the following fields:

- Tag
- Component
- Timestamp
- Content

In addition to 'tag' and 'component', each message contains a timestamp with a precision of milliseconds and the content, a string that can carry any data associated with the message. It can be left empty very often, as the generation of a message with this tag from a certain component already contains all necessary information. It is recommended to define a structure of the content field for each tag. In our example above this may be the name of the client for the 'beginRequest' tag and the empty string for the 'endRequest' tag.

2.1.4 Filters

Filters can be directly derived from the classification fields. Each filter consists of two sets. One set contains all tags that are of interest, the other one contains all components that are of interest. A special element (the character *) that matches all components/tags exists for each set. Through the combination of several filters, almost any subset of messages can be selected, allowing tools to define exactly the view they want.

2.2 The statemanager

Many tools are interested in the state of the monitored system. This state is represented by a number of messages in LogService. The problem is that some of these messages are generated only once at the startup of the system. As tools can connect at any time, they will not necessarily receive these messages.

A simple solution to this problem is to store all received messages and to resend them to connecting tools. This solution works, but it is not very efficient as most messages contain no information relevant for the system state. LogService refines this solution by identifying messages that affect the system state. These messages will be stored for resending while all other messages are just passed through. An advantage of this solution is that tools do not have to treat the system state separately. They always receive a stream of messages that represents the whole system after they connected.

This concept is implemented in LogCentral's statemanager. It identifies important messages by their tag and stores them for connecting tools. To further increase the performance, the statemanager knows several classes of important messages.

- Administrative messages.
  This group contains two tags *IN* and *OUT* which are automatically generated by the LogCentral if a component connects or disconnects. All messages with an *IN* tag will be kept until an *OUT* is received. This *OUT* message will not be stored. Instead all messages concerning it's component will be removed from the statemanager. This group is fixed and cannot be changed.

- Static messages and unique messages.
  Tags that are declared as unique or static will be stored until the component is disconnected with an *OUT*. The difference between unique and static is that only one message can exist for each unique tag, while several messages can exist for each static tag. This means that all messages with static tags will be stored, but messages with unique tags will overwrite older messages with this tag.

- Dynamic messages.
  Dynamic tags try to model processes that are started and stopped within a component. They always come in pairs. One tag indicates that the service identified by this tag is started, the second tag indicates that this service is stopped and will remove itself and the corresponding start message from the statemanager. This class can not reflect dynamic processes (yet) as only one process can exist for each tag. We hope that this problem will be solved in future versions of LogService.

2.3   Forwarding

The forwarder system is based on the same idea as the DIET forwarder. The main idea is simple, the forwarder manage a SSH tunnel between the two sides of the firewall, and resolves using object names the naming service reference to use. Thus, the user/programmer does not see any difference, he keeps thinking he is using its object directly, whereas on one side he is just using its name, and on the other side the name is resolved and the object is manipulated.

Here is a simple example of how to launch the forwarder between two machines:

- On the distant machine graal (graal.ens-lyon.fr)
  ./logForwarder –name ¡name_d¿ –net-config ¡path_d¿/forwarder.cfg

- On the local machine
  ./logForwarder -C –peer-name ¡name_d¿ –ssh-host graal.ens-lyon.fr –remote-port 50005 –name ¡name_l¿ –net-config ¡path_l¿/forwarder.cfg –remote-host localhost –ssh-login ¡login¿

Explanations

- logForwarder: Name of the forwarder programm
- –name ¡name_d¿/¡name_l¿: Name of the forwarder (available in the omninames after). name_d for distant name to give, name_l for local name to give.
- –net-config: Configuration file for the forwarder, they are called ¡path_d¿/forwarder.cfg and ¡path_l¿/forwarder.cfg in the example.
- -C: Option to create the SSH tunnel between the forwarder. It requires 3 more arguments :
  - –peer-name: Name of the corresponding forwarder on the other side (the name given to –name on the other side)
  - –ssh-host: Host to establish the SSH tunnel
  - –remote-port: A free port to create the SSH connection. He can be any free port (around 50000 may be a good idea to find a free one).
- –remote-host localhost: Option to add for loopback (otherwise there might be ssh problems)
- –ssh-login: Option if the login name on the other machine is different of the local login

3 Installation of LogService

3.1 Compiling the sources

The LogService sources can be obtained at `http://graal.ens-lyon.fr/DIET/download.html`. In order to use and install the package, omniORB 4 must be installed. The OmniORB package can be downloaded at `http://omniORB.sourceforge.net`. Please unpack, compile and install omniORB before proceeding. After that please unzip and untar LogService. You will find the following directories:

src/	All sources needed to compile LogService, the forwarder and the libraries.
src/idl/	All idl interfaces used by LogService and forwarder.
src/utils/	General classes that are not LogCentral specific.
src/monitor/	The main monitor program.
src/tester/	A small program that tests the monitor's internals. DEPRECATED
src/libraries/	The supporting libraries in C++ and java.
src/examples/	Two examples using the C++ libraries. The two tested tools are 'testTool' that creates a small tool displa

LogService compilation process moved away from the traditional `autotools` way of things to a tool named `cmake`. LogService requires using `cmake` at least version `2.4.3`. For many popular distributions `cmake` is incorporated by default or at least `apt-get` (or whatever your distro package installer might be) is `cmake` aware. Still, in case you need to install an up-to-date version `cmake`'s official site distributes many binary versions (alias packaged as tarballs) which are made available at `http://www.cmake.org/HTML/Download.html`. Optionally, you can download the sources and recompile them: this simple process (`./bootstrap; make; make install`) is described at `http://www.cmake.org/HTML/Install.html`

Compilation using cmake for the impatients : Assume that `LOGSERVICE_HOME` represents a path to the top level directory of LogService sources. This LogService sources directories tree can be obtained by users by expanding the LogService current source level distribution tarball. Additionally, assume we created a build tree directory and `cd` to it (in the example below we chose `LOGSERVICE_HOME/build` as build tree, but feel free to follow your conventions):

- `cd LOGSERVICE_HOME/build`
- `ccmake ..` to enter the GUI
  - press `c` (equivalent of bootstrap.sh of the autotools)
  - specify the `CMAKE_INSTALL_PREFIX` parameter (if you wish to install in a directory different from `/usr/local`,
  - press `c` again, for checking required dependencies
  - check all the parameters preceded with the * (star) character whose value was automatically retrieved by `cmake`.
  - provide the required information i.e. fill in the proper values for all parameters whose value is terminated by NOT-FOUND
  - iterate the above process of parameter checking, toggle/specification and configuration until all configuration information is satisfied
  - press `g` to generate the makefile
  - press `q` to exit ccmake
- `make` in order to classically launch the compilation process
- `make install` when installation is required

  The main configuration flag is :

- `OMNIORB4_DIR` is the path to the omniORB4 installation directory (only relevant when omniORB4 was not installed in /usr/local). Example: `cmake .. -DOMNIORB4_DIR:PATH=$HOME/local/omniORB-4.1.2`

  The install directory will contain the following subdirectories:

bin/	The LogCentral executable and a sample configuration file.
lib/	The libraries and their headers for java.

3.2 Quick start

To start a quick demonstration without forwarder, please run the following programs in the given order:

- *omniNames*. The CORBA Namingservice is needed by LogService. Please make sure that the namingservice is declared as an initial reference in the omniORB configuration file.

- *bin/LogCentral*. The *config.cfg* file will be selected by default.

- *examples/testTool*. This tool will display all messages. The connection of the tool should already be displayed by LogCentral.

- *examples/testComponent*. This program simulates a (very simple) component that just sends two messages. These messages can be watched with the testTool.

3.3 Configuration file

Before usage, several parts of the LogCentral should be configured by the user. This mainly includes the configuration of the statemanager and some internal parameters of LogCentral. All important options can be set in the mandatory configuration file that can be stored anywhere on your system. LogCentral knows two possibilities to define the location this file.

1. Add *-config pathToConfig.file* when calling LogCentral.

2. Set the environment variable *$LOGCENTRAL_CONFIG*.

If no config file is set, LogCentral tries to open the default config file *./config.cfg*.

The configuration file is composed of five sections. Each section begins with its [sectionname] in square brackets, followed by it's parameters. The sections are mandatory, even if they contain no parameters. Parameters are optional and only one parameter can be specified per line. Empty lines are ignored and comments start with '#' and stop at the end of the line.

The first section is the [general] section which allows the configuration of parameters that control the behaviour of LogCentral. It offers the following parameters:

- port = xxx
  Tells omniORB to use the port xxx for communication.

- MinAge = xxx (in milliseconds)
  Defines the minimum period of time that messages will be stored in the LogCentral. Messages that arrive out of order during this period will be sorted by the LogCentral. A high value guarantees correct sorting even under bad circumstances while a low value does not slow down the flow of the messages through the core.

- DynamicStartSuffix=START
  See [DynamicTagList]

- DynamicStopSuffix=STOP
  See [DynamicTagList]

All remaining sections configure the statemanager. They are named [DynamicTagList], [Static-TagList], [UniqueTagList] and [VolatileTagList] and correspond directly to the classes of the statemanager. The [VolatileTagList] section should contain all existing tags are not important. It is not essential for operation, but it can be requested by tools that want to find out which tags can be monitored.

All entries in the statemanager sections will be directly considered as tags. The only difference is the list of unique tags. The pairs cannot be defined freely but every tag in this list will be expanded with two suffixes to generate the pair. The default suffixes are *START* and *STOP*. So for example the tag *WORK* will be expanded to *WORK_START* and *WORK_STOP*. The names of the suffixes can be configured in the [General] section.

A correct configuration file could look like the following example.

```
[General]
port = 4242      # use a specific port
MinAge = 100     # 100 ms buffer time. This is not much, but
                 # it guarantes fast message forwarding
```

```
[DynamicTagList]
# we have no dynamic tags but the section is mandatory
# (even if empty)

[StaticTagList]
DESCRIBE_DATA

[UniqueTagList]
DATA_COUNT         # new DATA_COUNT messages will overwrite old ones

[VolatileTagList]
ADD_DATA
REMOVE_DATA
```

3.3.1 Forwarder configuration file

```
# accept everything from everyone
accept:.*


# reject nothing
reject:192\.168\.1\.6
```

The reject line corresponds to the IP of the machine on the network. It is safer than using reject:localhost which may make the messages to be misrouted.

3.4 Commandline arguments

LogCentral was designed to need as few commandline arguments as possible. Anyway it supports the full set of omniORB commandline options to change the behaviour of the ORB. Please read the omniORB manual for details.

4 How to connect to the LogCentral

4.1 Connection of components

This section documents how components can connect to the LogService. It explains the various interfaces and data types defined in the LogCentral IDL files and it describes how to use the LogComponentBase libraries to instrument your application easily.

4.1.1 The IDL interface

Let's first look at the IDL files. *LogTypes.idl* defines all data types that are used in the system, in particular the log_msg_t. All interfaces that must be known to connect a component to the LogCentral are define in *LogComponent.idl*. It's two big interfaces ComponentConfigurator and LogCentralComponent will now be described in detail.

```
interface ComponentConfigurator {
  void setTagFilter(in tag_list_t tagList);
  void addTagFilter(in tag_list_t tagList);
  void removeTagFilter(in tag_list_t tagList);
}
```

The ComponentConfigurator is an interface that must be offered by the component. This is necessary to allow the prefiltering of messages on the component. The filter on the component will be updated actively by the LogCentral whenever necessary by using this interface.

A filter on the level of the component (also called the component's configuration) is just a set of tags. It is represented by a list of strings in LogService that has the type `tag_list_t`. A new configuration for the component is passed with every call to the `ComponentConfigurator`. Depending on which method is called, the component must change it's filters accordingly. `SetTagFilter()` defines the whole filter. Each tag in it's list must be sent, all other tags must not be sent. `AddTagFilter()` and `removeTagFilter()` just modify the existing filter by adding or removing tags, leaving the status of all other tags untouched.

```
interface LogCentralComponent {
  short
  connectComponent(inout string componentName,
                   in string componentHostname,
                   in string message,
                   in string compConfigurator,
                   in log_time_t componentTime,
                   inout tag_list_t initialConfig);

  short
  disconnectComponent(in string componentName, in string message);

  oneway void
  sendBuffer(in log_msg_buf_t buffer);
[...]
```

The connect, disconnect and send function are the essential parts of the LogCentralComponent. Each component must connect to the LogCentral with `connectComponent()` to register itself and negotiate it's name. The component can propose a unique name or completely rely on autogenerated names. If the connect is successful, the tool will receive a unique name that it must use directly or indirectly for all following calls. The name will be valid until `disconnectComponent()` is called. This unregisters the component and frees it's name. Do not use the name any more after the disconnection.

The parameters of `disconnectComponent()` and `sendBuffer()` should be self explaing, but the signature of `connectComponent()` is complicated and will thus be explained in detail.

- `componentName`
  The name of the component. If this name is empty, the LogCentral will create a unique name for this component and return it in this variable. Otherwise the LogCentral will check if this name is already connected. It will return with LS_COMPONENT_CONNECT_ALREADYEXISTS immediately if the component could not be registered.

- `componentHostName`
  The name of the component's host. Autogenerated names will be based on this name to allow rudimentary identification of autogenerated components. LogCentral does not rely on this parameter.

- `message`
  The message that will be sent with the *IN* message that will be generated by LogCentral.

- `compConfigurator`
  A string corresponding to the name of the component to be called. It is a string and not a CORBA reference since the forwarders cannot use references directly.

- `time`
  The actual time on the component right before connection. This value is used to determine the initial latency between component and monitor.

- `initialConfig` (OUT)
  The initial filter configuration of this component. The LogCentral might also use the ComponentConfigurator, but passing this parameter as an out parameter guarantees synchronisation from the beginning.

```
[...]
  void
  ping(in string componentName);

  void
  synchronize(in string componentName, in log_time_t componentTime);

  void
  test();
}
```

The remaining functions of the interface deal with the time synchronisation and the heart-beat of components. The LogCentral automatically disconnects inactive components, so each component has to call `ping()` in regular intervals keep the connection alive. The `synchronize()` function recomputes the latency between the component and the monitor. It also should be called from time to time to react upon changes in the network connection. `Test()` exists to control the status of a LogCentralComponent reference. It does nothing and can be used to check if the reference is valid.

Components that want to connect to the LogCentral need to obtain a reference to it's `LogCentralComponent` servant. This reference will be registered in the Naming Service by Log-Central. The name of the reference is 'LCC' in the context 'LogServiceC'. The kind of reference and context is empty.

4.1.2 The LogComponentBase library

DEPRECATED (the library is out of date)

All the interfaces described above are implemented in the LogComponentBase library. It hides many details of the connection and the ComponentConfigurator and can be used to easily create a component for a certain application. Using the LogComponentBase requires only the most essential parameters for each function. The name of the component can be set with `setName()`, connection and disconnection just require the message and all synchronisation is done internally. The application can use the `isLog()` function to find out if a certain tag is required in the moment or not. Messages can easily be generated by `sendmsg()`.

As the LogComponentBase is available in C++ and Java, no exact description of the class is given here. Look at the C++ header file[1] and the java source code[2] for details. The C++ library is based on omniORB and will be built together with the LogCentral. The java library relies on JAVAs built in ORB and is distributed in source code only. There is no compilation support through the makefiles, so you must compile the IDL files and all the java classes yourself. Please note that the LogComponentBase was not extensively tested yet and may still contain bugs.

4.2 Connection of tools

4.2.1 The IDL interface

This section explains how tools can connect to the LogCentral. It presents the interfaces that can be used to define filters and to receive messages as well as the LogToolBase, a library that can be used to easily build tools.

The essential interfaces to connect tools to the LogCentral are defined in *LogTools.idl*. It relies on *LogTypes.idl* just like *LogComponent.idl* and it's main interfaces will be presented now.

```
interface ToolMsgReceiver
{
  oneway void
  sendMsg(in log_msg_buf_t msgBuf);
};
```

---

[1] *libraries/LogComponentBaseC++/LogComponentBase.hh*
[2] *libraries/LogComponentBaseJava/LogComponentBase.java*

Whenever the monitor receives a message that is of interest for a tool, it actively forwards this messages to the tool. This is done through the `ToolMsgReceiver`, an interface that must be implemented by each tool. It offers just one function that allows the receiving of messages. If several messages are sent at once, then the earliest message will be at the beginning of the buffer.

```
interface LogCentralTool
{
  short
  connectTool(inout string toolName, in ToolMsgReceiver msgReceiver);

  short
  disconnectTool(in string toolName);

  short addFilter(in string toolName, in filter_t filter);
  short removeFilter(in string toolName, in string filterName);
  short flushAllFilters(in string toolName);

  [...]
```

The `LogCentralTool` interface is very close to the `LogCentralComponent` interface. Like components, tools receive a unique name upon connection. This name serves as an identifier for all future calls and can either be set by the tool or generated by the LogCentral. Upon connection, a tool must pass a reference to it's `ToolMsgReceiver` to be able to receive messages. After that, it can define it's filters using the `addFilter()` function. The `filter_t` is composed of a list of tags, a list of components and a unique name for the filter. As this name must only be unique within the scope of this tool, no mechanism to automatically generate it is provided. Each tool must ensure itself that it does not define two filters with the same name. If this happens anyway, LogCental will deny to add this filter by returning `LS_TOOL_ADDFILTER_ALREADYEXISTS`.

`RemoveFilter()` and `flushAllFilters()` do exactly what their names propose. They remove one or all defined filters of this tool.

```
  [...]

  tag_list_t
  getDefinedTags();

  component_list_t
  getDefinedComponents();

  void
  test();
```

The remaining functions of the `LogCentralTool` interface mainly provide convenience services. `Test()` can be used to check the validity of a LogCentralTool reference. `GetDefinedTags()` returns the list of available tags as defined in the configuration file. Tools who are written to monitor a certain application usually don't need this information, but there may exist general purpose tools which are interested in which tags are available. `GetDefinedComponents()` returns a list of currently connected components. A tool can compute this list itself by processing the received *IN* and *OUT* messages, but very simple tools might want to use this function for convenience reasons.

Tools that want to connect to the LogCentral need to obtain a reference to it's `LogToolComponent` servant. This reference will be registered by LogCentral in the Naming Service. The name of the reference is 'LCT' in the context 'LogServiceT'. The kind of reference and context is empty.

4.2.2 The LogToolBase library

DEPRECATED (the library is out of date)

Just like the LogComponentBase, the LogToolBase hides many details of the connection and the message receiving. In order to use the LogComponentBase, the user must inherit from it and

overwrite the abstract function sendMsg(). This function will be called by the LogComponent-Base's MsgReceiver if a message is received.

A C++ and a Java implementation of the LogToolBase exist. Both follow the interface defined in the IDL quite straightforward and will not be explained here. Look at the C++ header file[3] or the java source code[4] for detail. The C++ library is based on omniORB and will be built with the LogCentral. The java library relies on JAVAs built in ORB and is distributed in source code only. There is no compilation support through the makefiles, so you must compile the IDL files and all the java classes yourself. Please note that the LogToolBase was not extensively tested yet and may still contain bugs.

5 How to write a quick component/tool

5.1 Component

An example of functionnal component can be found in DIET, in DIET_HOME/src/utils/log/DIETLogComponent.cc. The defined component must inherit from the component configurator IDL (the POA_ComponentConfigurator class) and have a Log Central Component attribute (type LogCentralComponent_var). The class must redefine 3 main functions plus some useful functions:

- The 3 main
  - void setTagFilter(const tag_list_t& taglist)
  - void addTagFilter(const tag_list_t& taglist)
  - void removeTagFilter(const tag_list_t& taglist)
- The useful (that are not to be called by the user)
  - void synchronize() : synchronization with the log central
  - void ping() : ping the log central

Moreover, the class must provide log function(s). For example, it may offer

```
void log (char* tag, char* msg){
  log_msg_t logM;
  log_msg_buf_t out;

  logMsg.componentName = CORBA::string_dup(name);
// name is the name of the component stored as an attribute of the class
  logM.time = getLocalTime();
  logM.tag = CORBA::string_dup(tag)
  logM.msg = CORBA::string_dup(msg)

  out[out.length()] = logM;
  LCC->sendBuffer (out); // LCC is the Log Central Component attribute
}
```

The initialisation of the LCC element is done with a LogORBMgr::getMgr()->resolve('LogServiceC', 'LCC') call, then you must bind (and fwdsBind) using _this() and the name as parameters. Finally you can connect the component with the connectComponent() call.

5.2 Tool

The examples testTool and DIETtestTool are 2 functionnal examples of how to make a tool.

---

[3]*libraries/LogToolBaseC++/LogToolBase.hh*
[4]*libraries/LogToolBaseJava/LogToolBase.java*

First the tool must inherit from POA_ToolMsgReceiver and have a LogCentralTool attribute. The initialisation of the log central tool attribute is made with a resolve ('LogServiceT', 'LCT'), then the tool is binded (with its name and _this()). Finally the tool can be connected with a connectTool call. A filter can be created and added with a addFilter call.

The class must only redefine the sendMsg(const log_msg_buf_t& msg) function.

Nothing more is needed to have a functionnal tool (in fact you need a main and call the initialisation sequence to have a functionnal tool)

6 Technical details of LogCentral

By now you should know everything you need know to use LogService. It is not mandatory to read this section as it mainly contains technical information on how LogCentral is implemented. It also explains LogCentral's main components and the way they work together. This section mainly addresses developers who want to change the sourcecode and give them a short overview of LogCentral's internals.

6.1 Overview of the log

Before explaining detailed scenarios we present the component diagram of LogCentral (see figure 1). Objects are denoted by boxes, fixed relationships by arrows and threads by circles. Please note that the the LogCentralComponent_impl and the LogCentralTool_impl are servants that can be called by the ORB while SendMsgReceiver and ComponentConfigurator are proxies for distant objects.

A very important part of the system is the FullLinkedList (FLL) template. It offers a double linked list that guarantees synchronisation. If an object wants to access the list, it has to acquire an iterator. This iterator guarantees synchronous operation by locking the FLL's internal read/write mutex. This mutex won't be unlocked until the list is destroyed. All synchronisation in LogCentral is done by using this mutex in the FLL.

The ToolList and the ComponentList contain data shared by all objects, the Options class acts as a central storage for all constants and the ReadConfig class is responsible for parsing the configuration file. All other classes will be explained in the following sections that represent the four major usecases of LogCentral.

6.2 Overview with the forwarders

The following image represent an overview of what needs to be running for the forwarder.

- In magenta, there are on each side of the network the omninames running.

- In yellow, there are the 2 forwarder, they communicate throught a ssh tunnel. Both of them are under the same naming context.

- In grey is the LogCentral, the 2 boxes inside represent its component and its tool he registers in the naming context.

- In orange are the tools, both of them are under the same context in the omninames and linked to the log central. The first one is direct (black link), the red one is virtual, in fact this tool is linked going through the forwarders, using the name saved in the omninames.

- In green are the components, both of them are under the same context in the omninames and linked to the log central. The first one is direct (black link), the red one is virtual, in fact this component is linked going through the forwarders, using the names saved in the omninames.
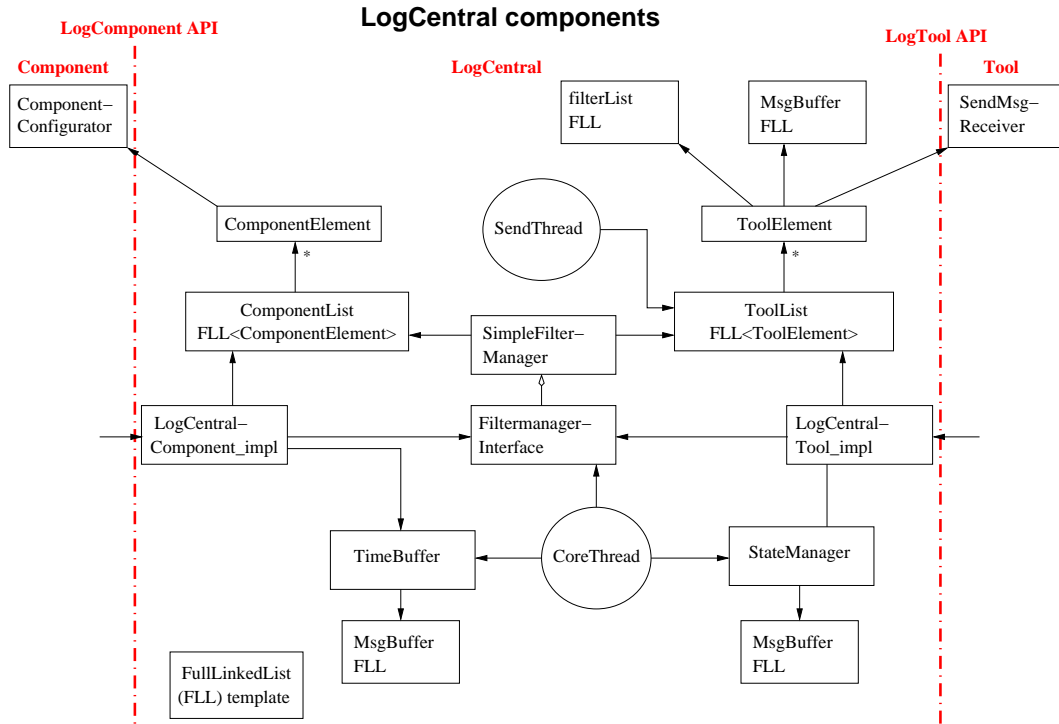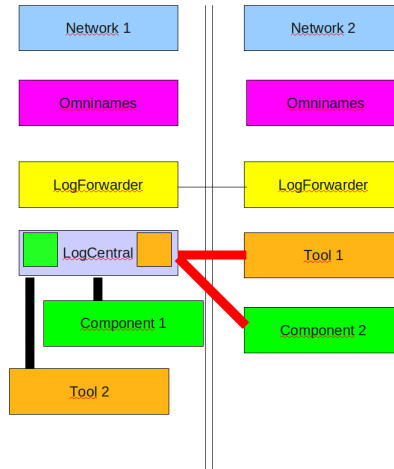
Figure 1: LogCentral components

Figure 2: Example using the forwarders

6.3 Connection of Components

A component connects through the appropriate methods in `LogCentralComponent_impl`. After checking it's name and generating a unique name if necessary, the `LogCentralComponent_impl`

inserts the component in the `ComponentList`. It then notifies the filtermanager through the `FilterManagerInterface` and receives the components initial configuration. After that it inserts the component in it's internal list that stores the latency and the last ping from the component. The time of the last ping will be checked regularly by the `AliveCheckThread`, an internal thread of the `LogCentralComponent_impl`. The Disconnection of components works just the same.

6.4 Connection of Tools

The connection of tools works like the connection of components. A tool connects through the `LogCentralTool_impl`. It is inserted in the `ToolList` and the `FilterManagerInterface` is notified. After that the `LogCentralTool_impl` asks the `StateManager` to copy the actual system state in the `OutBuffer` that has been assigned for the new tool. For disconnection, the `FilterManagerInterface` is notified and the corresponding `ToolElement` is removed from the list. Remaining messages and filters are deleted by this.

6.5 Configuration of filters

Filters will be configured through the `LogCentralTool_impl`. It checks the name of the tool as well as the name of the filter, inserts the filter in the tool's `FilterList` and notifies the `FilteManagerInterface`. The filtermanager will calculate a new configuration for each component. It will check for each component if the configuration has changed and (if necessary) forward the new configuration through the `ComponentConfigurator` assigned to this component.

6.6 Processing messages

Messages enter the system at the `LogCentralComponent_impl` where their timestamp is altered according to the internal latency list. Then the messages are stored in the `TimeBuffer` one by one. The timebuffer will sort the messages automatically in it's internal list. They remain there until the `CoreThread` checks the timebuffer to find messages that are older than a defined minimum age. It will remove the oldest message and check with the help of the `StateManager` if this message is important for the systemstate. If yes, the statemanager will keep a copy of this message in it's statebuffer and the corethread will broadcast this message to all tools. Otherwise the corethread passes the message to the `FilterManager` which will apply the necessary filters for each tool and forward the message to all interested tools. In both cases the message is not sent directly, but put in the tool's `OutBuffer`. The outbuffers are cleared regularly by the `SendThread`. It checks each tool's outbuffer and uses the `ToolMsgReceiver` to actually send the messages.

7 Future works

LogService was written to monitor the DIET system. As such the project is finished and will hopefully work properly the next years. Anyway some parts of LogService could be enhanced. First of all the `SimpleFilterManager` could be replaced by filtermanager with better reconfiguration algorithms to increase the overall performance of the system. Second the statemanager should eventually be changed to allow a more sophisticated description of corresponding messages that allow a better model of the systemstate.

If you want to participate in this work or if you have any other questions concerning LogService, don't hesitate to mail the authors[5] or the graal staff[6]. Any kind of feedback will be appreciated to ensure the future development of this software.

---

[5] Georg Hoesch: hoesch@in.tum.de, Cyrille Pontvieux: cyrille.pontvieux@edu.univ-fcomte.fr
[6] Please follow the links on `http://graal.ens-lyon.fr`