Lire Developer’s Manual

Joost van Baal

Egon L. Willighagen

Francis J. Lacoste

Lire Developer's Manual

by Joost van Baal, Egon L. Willighagen, and Francis J. Lacoste

Revision History

Revision 1.1 $Date: 2002/08/18 23:37:45 $
$Id: dev-manual.dbx,v 1.55 2002/08/18 23:37:45 flacoste Exp $

Table of Contents

List of Figures

List of Examples

Preface

Log file analysis is both an essential and tedious part of system administration. It is essential because it's the best way of profiling the usage of the service installed on the network. It's tedious because programs generate a lot of data and tools to report on this data are often unavailable or incomplete. When such tools exist, they are generally specific to one product, which means that you can't compare e.g. your Qmail and Exim mail servers.

Lire is a software package developed by the Stichting LogReport Foundation to generate useful reports from raw log files of various network programs. Multiple programs are supported for various types of network services. Lire also supports various output formats for the generated reports.

What This Book Contains

This book is the *Lire Developer's Manual*. Its purpose is to present Lire as a log analysis framework. To this ends, it describes the architecture and design of Lire and contains comprehensive instructions on how to use it. Its intended audience is system administrators or programmers who want to extend Lire or want to understand its internals.

There is another book, the *Lire User's Manual* which describes how to install, configure and use Lire, as a "off-the-shelf" log analyzer. Its intended audience is system administrators who want to install and use Lire to gather information about the services operating on their network.

How Is This Book Organized?

This book is divided in five parts. Part I gives an overview of the architecture and design of Lire.

You will find in Part II information on extending Lire. In this part, you will learn how to add a new DLF format to Lire, write log file converters and add reports for a superservice.

Part III is a reference section which gives comprehensive details about the various XML formats used by Lire and gives in-depth descriptions of its various APIs.

Part IV is targeted at developers who want to participate in Lire's development. It contains information about CVS access, coding conventions, tools needed to build from CVS, release management and other aspects important to those part of the Lire development team. Furthermore, it gives some information on how to contribute code to Lire, as an external party.

Finally, Part V contains various implementation details that may be interesting to people wanting to learn more about Lire internals.

Conventions Used

If You Don’t Find Something In This Manual

You can report typos, incorrect grammar or any other editorial problems to <bugs@logreport.org>. We welcome reader’s feedback. If you feel that certain parts of this manual aren’t clear, are missing information or lacking in any other aspect, please tell us. Of course, if you feel like writing the missing information yourself, we’ll very happily accept your patch. We will make our best effort to improve this manual.

Remember, that there is another manual, the *Lire User’s Manual* which contains comprehensive information on how to install, use and configure Lire. It also contains reference information about all of Lire’s standard reports and supported services.

There are various public mailing lists for Lire’s users. There is a general users’ discussion list where you can find help on how to install and use Lire. You can subscribe to this list by sending an empty email with a subject of *subscribe* to <questions-request@logreport.org>. Email for the list should be sent to <questions@logreport.org>.

You can keep track of Lire’s new release by subscribing to the announcement mailing list. You can subscribe yourself by sending an empty email with a subject of *subscribe* to <announcement-request@logreport.org>.

Finally, if you’re interested in Lire’s development, there is a development mailing list to which you can subscribe by sending an empty email with a subject of *subscribe* to <development-request@logreport.org>. Email to the list should be sent to <development@logreport.org>.

All posts on these lists are archived on a public website.

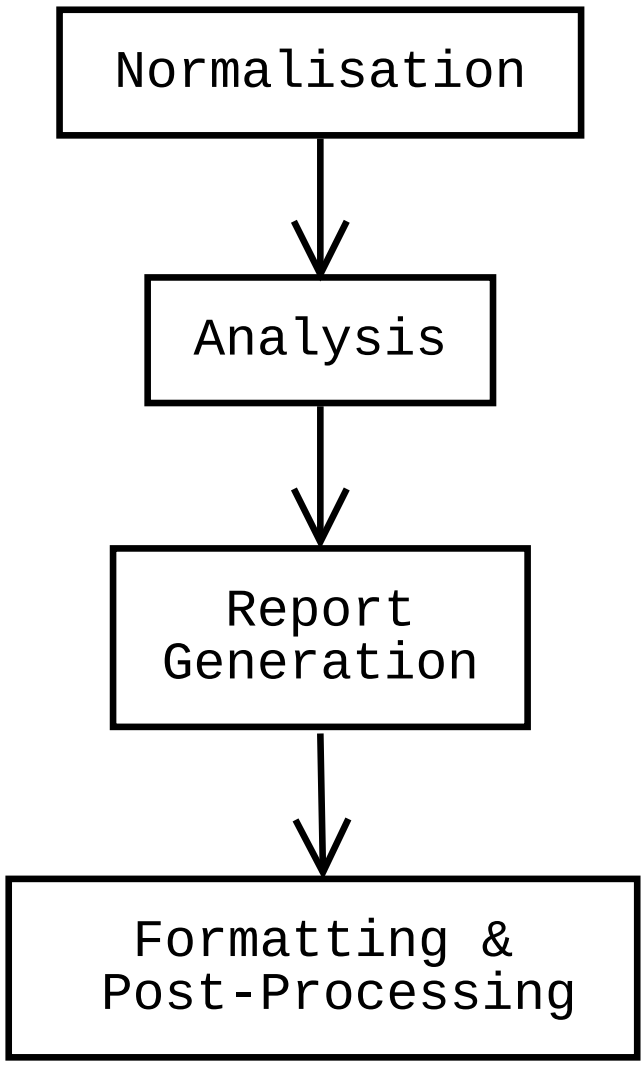
I. Lire Architecture

Chapter 1. Architecture Overview

From a developer's point of view, Lire intends to be the universal log analysis framework. To this end, it provides a reliable, complete, framework upon which to build log analysis and reporting solution. Lire, the tool, is a proof of the versality and extendability of the framework as it is able to produce reports for many of the services that run in today's heterogeneous networks in a variety of output formats.

As a framework, Lire is the best choice to replace all those home-grown scripts developed to produce reports from all the log files from the little-known products or custom-developed programs that run on your system. Leveraging Lire framework will make those scripts a lot more versatile while not being really more complicated to develop. It will be easier to add new reports or to support multiple report formats.

Figure 1-1. Log Processing in the Lire's Framework



The Lire's framework divides log analysis in four different processes. The figure Figure 1-1 shows those four processes:

1. **Log Normalisation.** The first process normalise logs from different products into a generic format that can be shared by all products that have similar functionality. For example, log files from products as different as Apache and Microsoft Internet Information Server will be transformed into an identical format.
2. **Log Analysis.** In the analysis process, other information is created, inferred or extracted from the normalised data. For example, an anlyser in the www superservice infers the browser used by the client from the referrer information.
3. **Report Generation.** The third process generates a report from the normalised and analysed data. This process is done by a generic report engine that computes the report based on specifications describing what and how the information should appear in the report. The report is generated in a generic XML format.
4. **Report Post-processing and Formatting.** The last process converts the generic report into a specific format like ASCII, PDF, HTML but other kind of post-processing (like charts generation) can also be accomplished in this stage.

Before going into a more detailed description of each of these procesesses, we'll introduce some of the common design's patterns that you'll find throughout the Lire's framework.

Lire's Design Patterns

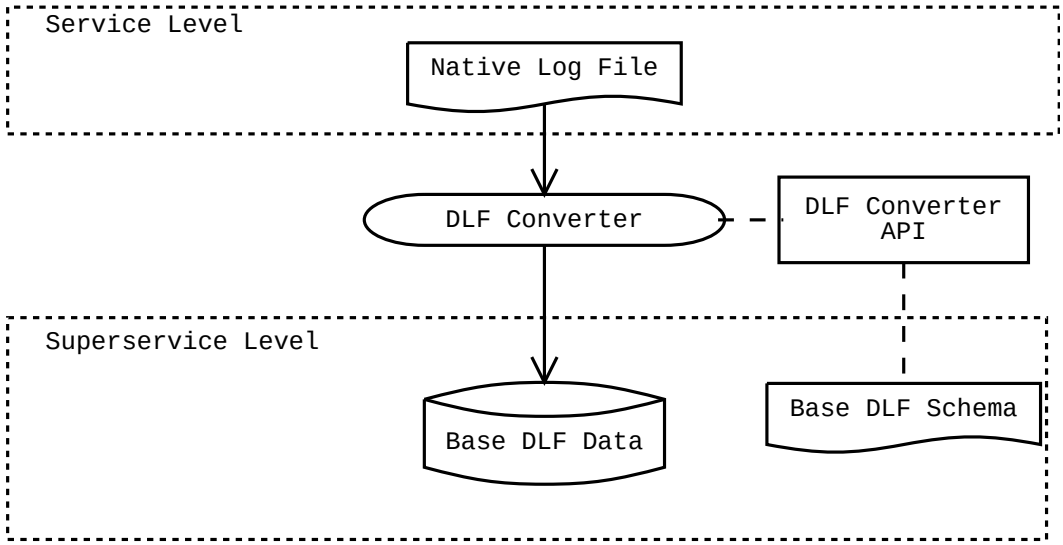
At the center of each of these processes is an XML based file format. Having things specified in data files makes it easier to extend. For example, the reports are built using a generic report builder which finds the instructions on how to build the reports in XML files. So this makes it easy to add new information to a report: you just have to write an XML file. The fact that there are a lot of tools to process XML files is also an interesting aspect. For example, emacs lovers will appreciate the help that its psgml module gives them in writing report specifications.

Another important aspects is that we tried to interoperate and to build upon other standards while defining our XML formats . The best illustration of this is that in all the XML file formats that Lire use, a DocBook subset is used for all elements related to narrative descriptions.

Another common aspect you'll encounter is that each of these processes and XML file formats come with an API to manipulate them, making it easy to add functionalities at each processing stage. APIs are also a good thing because, even if in theory an open file format somewhat constitutes an API, having libraries that provide convenient access to the file formats makes it a lot easier to write new components providing new functionalities.

Log File Normalisation

Figure 1-2. The Log Normalisation Process



The first process of the Lire log analysis framework is the log file normalisation process. That process is summarized in the Figure 1-2 figure. This process is centered around the *DLF* concept which is kind of a universal log format. DLF stands for Distilled Log Format. The concept is that each product specific log file is transformed into a log format that can be common to all the products providing similar functionalities. In Lire's terminology, a class of applications providing similar functionality (e.g. MTA's supplying email) is called a *superservice*. Still in Lire's terminology, the *service* from which the super is derived (e.g. postfix or sendmail) refers to the native log format that is converted in the superservice's DLF. One can view the DLF as a table where the rows are the logged events and the fields are logged information related to each event.

Since the information logged by an email server is totally different from a web server, each superservice should have its own data models. In Lire, the data model is called a DLF *schema*. The DLF schemas are defined in XML files using the DLF Schema Markup Language. The schema describes what fields are available for each logged events.

One interesting aspect of Lire, is that altough the email DLF is used by all email servers, the email DLF data model isn't restricted to the lowest common denominator across the log formats supported by each email servers. In the Lire's architecture, the superservice's schema can represent the information logged by the most sophisticated product. When some part of the information isn't available in one log format, the DLF log file will contain this information and the reports that needs this information won't be included.

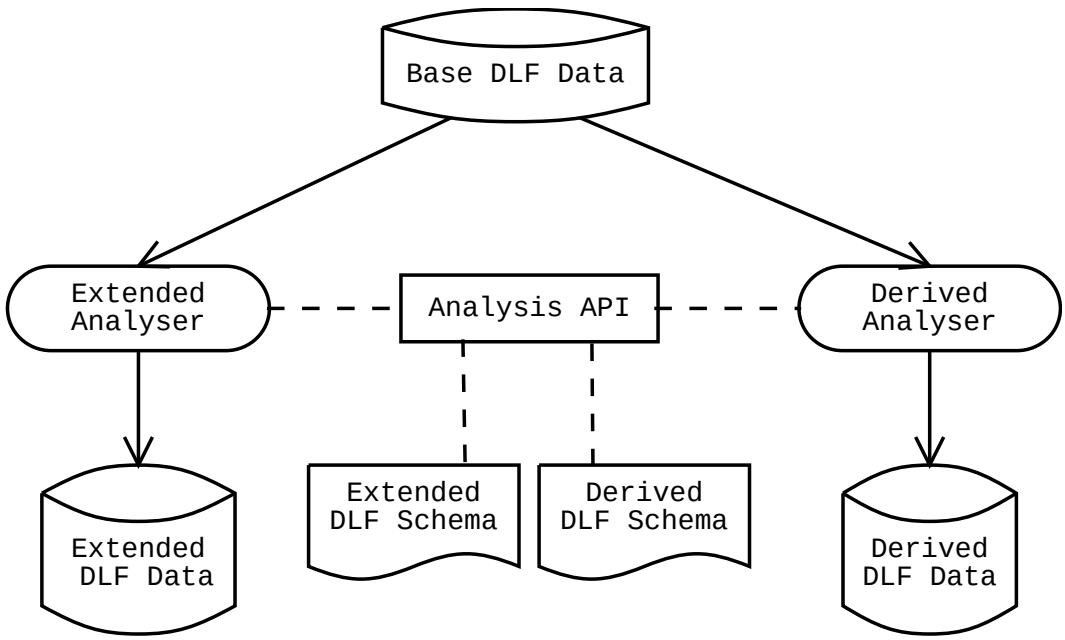
This architecture means that to support a new service, i.e. a new log format, in Lire you just need to write

a plugin, called a DLF converter. This is just a simple perl script that parses the native log format and maps the information according to the schema.

Log Analysis

After normalisation, comes the analysis process. The analysis process responsability is to extracts, infers or derives other information from the logged data. Since the superservice's logged data is in a standard format, the analysers are generic in the sense that they can operate for all the superservice's supported log formats, if the product's was clever enough to log the information required by the analyser. The analysis process is shown in the Figure 1-3 figure.

Figure 1-3. The Log Analysis Process



Since each analyser can add information to or create a new DLF, each analyser will generate data according to special kind of schemas.

Lire's framework include two kind of analysers. The difference between the two resides in the mapping between the source data and the new data they generate. Extended analysers generate new data for each DLF record whereas derived analysers are used when the new data doesn't have a one-to-one mapping with the source data.

The analysers produce data according to a data model which is specified in other DLF schemas. There are *extended* schemas and *derived* schemas. An extended schema simply adds new fields to the base

superservice's schema. For example, in the web superservice's schema, a lot of information can be obtained from the referer field. From this information, it is possible to guess the user's browser, language or operating system. Those fields are specified in the www-referer extended schema; one analyser is responsible for extracting this information from the referer field.

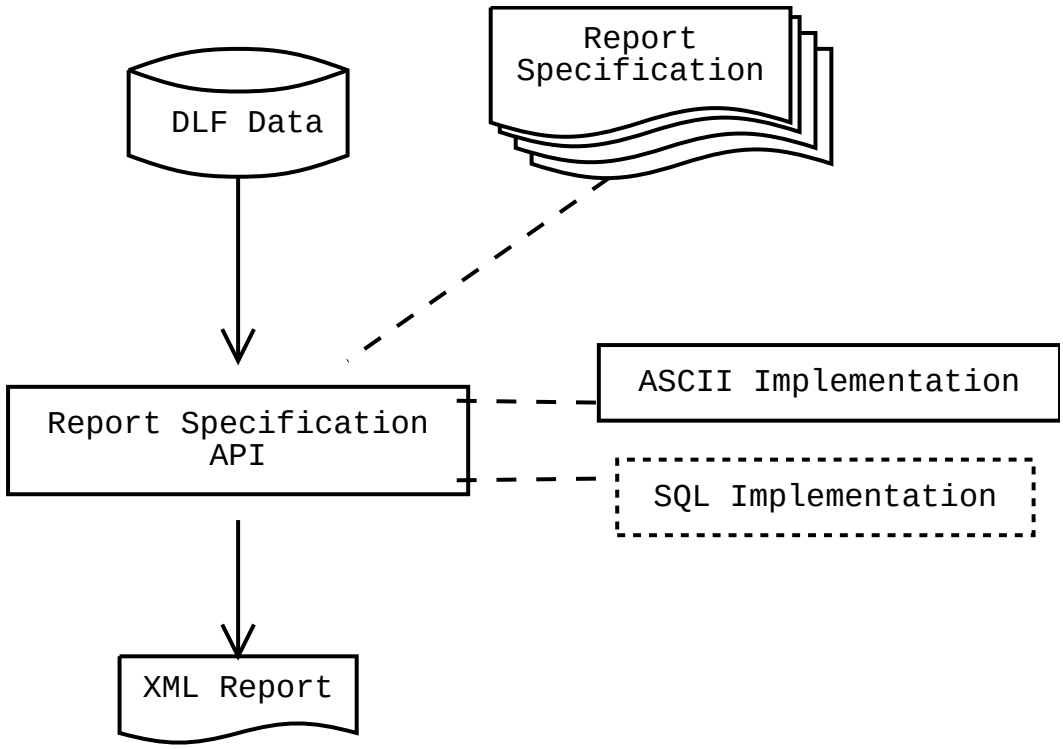
But sometimes the analysis cannot just simply add information to each event record, an altogether different schema is needed then. For those cases, there is the derived schema. An example of the use of such a schema in the current Lire distribution is the analyser which creates user sessions based on the logged client IP address and user agent. This analyser defines the www-session derived schema.

Analysers are simple perl modules that receive the base superservice's DLF records and output DLF records in the extended or derived schema. The architecture supports cascading of schemas; this feature isn't used anywhere now.

Report Generation

Once you have all this data, it's time to generate some useful reports out of it. Lire's framework includes a generic report builder. What Lire calls a *report* is actually a *collection* of what one may understand as reports; Lire however speaks about a *subreports*. For example, the proxy's superservice report will contain subreports about the top visited sites, another subreport on the cache hit ratio, as well as several others. The subreports are defined using the *Report Specification Markup Language*. This markup language contains elements for several things: information regarding the schema on which it operates; descriptions that should be included in the generated report to help in the interpretation of the data; parameters that can be used to modify the generated report (for example, to generate a top 20 subreport instead of a top 10); a filter that selects the records that will be used for the subreport; and finally the operations that make up the subreport: grouping, summing, counting, etc. The report markup language covers most simple needs and there is an extension element as well as an API that can be used to hook in more fancy computations. There are no subreport specifications in the current distribution that make use of this feature yet, however. You can see an overview of this process in the Figure 1-4 figure.

Figure 1-4. Report Generation Process



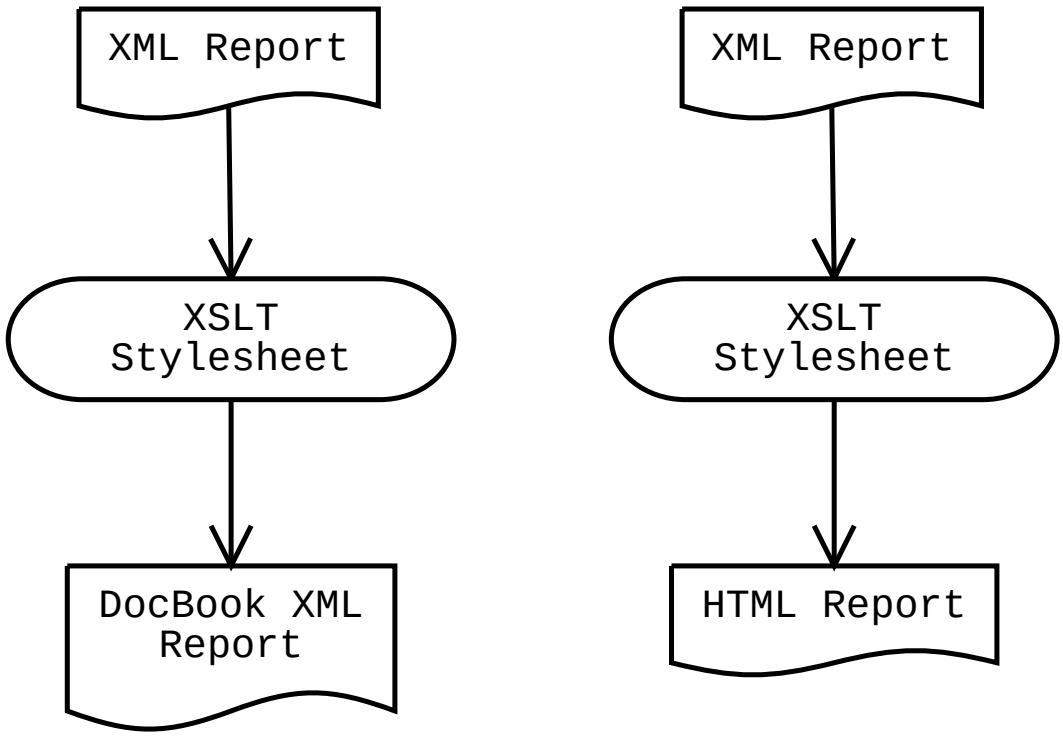
The actual computation of the subreport is delegated to another module of the framework. Presently, the DLF concept is implemented as simple space delimited log files. This makes it very portable and very simple, but for huge amounts of log data this isn't necessarily the best solution. But it would be easy to switch to a database driven backend where the DLF records are hold in tables and the subreport specifications are mapped to SQL queries.

The generated report is another XML file that uses another markup language, this time called the Lire's Report Markup Language. An actual report contains the help descriptions from the report specifications, information on the subreport specifications used, as well as the actual subreport's data.Using another intermediary XML file as output format makes all sort of things possible in the formatting and post-processing stage.

Report Formatting and Other Post-Processing

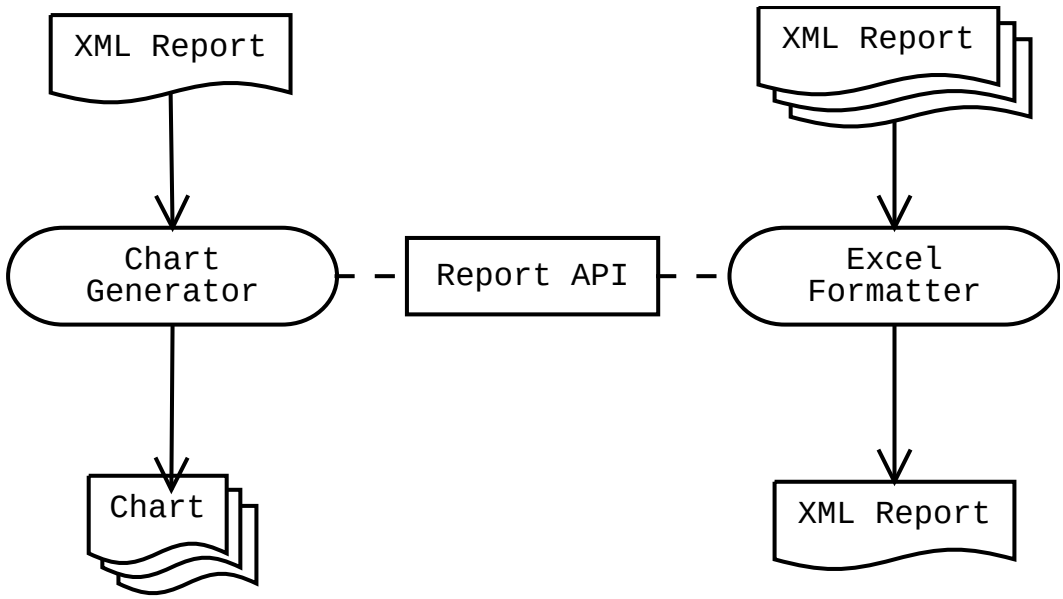
The last process works with the generic XML report. Using a domain-specific XML format for the generated format makes it easy for the framework to support multiple different formats. Supporting a new output format is just a matter of writing a new module that processes the XML report file.

Figure 1-5. XSLT Processing of the XML Report



There are many ways to do this. Of course, there is the XSLT way which can be used to format the report using stylesheets to convert to other XML formats. That’s how we support the HTML, XHTML, LogML and PDF formats. Figure Figure 1-5 gives an example of such processing.

Figure 1-6. Processing of the XML Report Using The APIs



As shown in the Figure 1-6 figure, you can also process the XML files using the APIs to the XML report format. That's how charts generation and the Excel backend are implemented. There's actually two APIs to process XML reports; one is event based and the other is object based. That's making different styles of programming possible. That's the equivalent of the SAX vs DOM model in the XML world.

Going Further

As you can see form this overview, the Lire framework provides a powerful architecture to use for your log analysis needs. The architecture provides extensibility from log normalisation to post-processing of the reports. Exactly how to use the framework is the topic of the next part.

II. Using the Lire Framework

Chapter 2. Writing a New Superservice

Writing a new superservice involves several things:

1. Making new directories in CVS:
   - `/service/<superservice>/`
   - `/service/<superservice>/script/`
   - `/service/<superservice>/reports/`
2. Adding several files:
   - `/service/<superservice>/Makefile.am`
   - `/service/<superservice>/reports/Makefile.am`
   - `/service/<superservice>/script/Makefile.am`
   - `/service/<superservice>/<superservice>.cfg`
   - `/service/<superservice>/<superservice>.xml` This file specifies the DLF format of the superservice. Ideally, it should offer a place for each and every snippet of information which will ever be found in a logfile from a program which offers functionality defined by the superservice. This file should have documentation embedded; this will show up in this manual.
3. Writing service plugins (2dlf scripts):
   - `/service/<superservice>/script/<service>2dlf.in`
4. Adapting several files:
   - `/service/configure.in` (add the Makefiles and 2dlf script to AC_OUTPUT, to get them converted from <service>2dlf.in to <service>2dlf.)
   - `/service/Makefile.am` (add the superservice directory to SUBDIRS, so that make gets run there too, when called from the root source directory.)
   - `/service/all/etc/address.cf` (to make the new service known as a member of a superservice.)
5. Update Documentation:
   - User Manual: Chapter "Supported Applications".
   - Add manpages for scripts
   - This document: add a referal to the superservice-schemas.dbx file, which gets build from superservice.xml.
   - The User Manual: add referals to superservice-reports-infos.dbx and superservice-filters-infos.dbx.
6. Update **lr_config**

DLF Design

The DLF Schema

Chapter 3. Writing New Service

Since Lire 1.1, adding a service is as simple as sticking your 2dlf script in `libexec/lire/convertors/` and registering it as belonging to a superservice in `address.cf`.

Writing a Log File to DLF Converter

API for 2DLF Scripts

Chapter 4. Writing a New Report

Writing a new report involves writing a report specification, e.g. `/service/<superservice>/reports/top-foo-by-bar.xml`, and adding this report along with possible configuration parameters to `<service>.cfg`. E.g., to create a new report, based upon `email/from-domain.xml`: copy the file `/usr/local/etc/lire/email.cfg` to `~/.lire/etc/email.cfg`. Copy the file `/usr/local/share/lire/reports/email/top-from-domain.xml` to e.g. `/usr/local/share/lire/reports/email/from-domain.xml`. Edit the last file to your needs, and enable it by listing it in your `~/.lire/etc/email.cfg`.

Beware! The name of the report generally consists of alphanumerics and '-', but the name of parameters may *not* contain any '-' characters. It generally consists of alphanumerics and '_' characters.

Report Information

Report's Display Specification

Filter Specification

For now, you'll have to refer to the example filters as found in the current report specification files. We'll give one other example here: specifying a time range.

Suppose you want to be able to report on only a specific time range. You could build a (possibly global and reused) filter like:

```
<filter-spec>
  <and>
    <ge arg1="$timestamp" arg2="$period-start"/>
    <le arg1="$period-end" arg2="$timestamp"/>
  </and>
</filter-spec>
```

Calculation Specification

Chapter 5. Writing Advanced Reports

Using a Derived Schema

Writing Extension Reports

III. Developer’s Reference

Chapter 6. Schemas Reference

This chapter documents the available schemas in the standard Lire suite. For each superservice, the base schema is explained, followed by any extended and derived schemas.

Schemas for the database Superservice

DLF Schema for Database service

Schema ID: `database`
Timestamp Field: `time`

A record in the database DLF schema represents one event in the database. This may be a connection from a client, a SQL query, etc.

Fields in the Schema

`time`

Type: timestamp
Defaults: 0

The time at which the event occured.

`user`

Type: string
Defaults: -

The name of the user who executed the command.

`remote_host`

Type: hostname
Defaults: -

The host from which the user executed the command.

`action`

Type: string
Defaults: -

The command that was executed. Commands you are likely to encounter are `connect`, `disconnect`, `shutdown` and `query`.

`database`

Type: string
Defaults: -

The database name on which the command was executed.

`query`

Type: string
Defaults: -

When the *`command`* field is a `query`, this field contains the actual SQL query that was executed.

`success`

Type: bool
Defaults: -

Did the command succeeded?

`result`

Type: string
Defaults: -

When the *`success`* field is `false`, i.e. when the command failed, this field should contains the logged error message.

`connection_id`

Type: int
Defaults: -

An "appropriate" "connection label" for the backend that can be used for session analysis in conjunction with the *`timestamp`*, *`username`* and *`database`* fields. This can be a connection identifier, a PID, a real session ID or whatever makes sense in the particular backend.

Extended Schemas for the database Superservice

Query Type Extended Schema for Database Superservice

Schema ID: `database-querytype`
Base Schema: `database`
Module: `Lire::Extensions::Database::DatabaseSchema`
Required Fields: query

An extended schema for the database superservice which extracts the query type from the query that was made.

Fields in the Schema

`querytype`

Type: string
Defaults: -

The type of SQL query that was made. This will be usually be something like `SELECT`, `INSERT`, `UPDATE`, `DELETE` or other administrative commands. In the case of nested queries, this will be the type of the outer-most query.

Schemas for the dialup Superservice

DLF Schema for Dialup service

Schema ID: `dialup`
Timestamp Field: `time`

The dialup DLF schema represents each connection attempt with one DLF record.

Fields in the Schema

`time`

Type: timestamp
Defaults: 0

The time at which the dial up connection started.

`local_number`

Type: string
Defaults: -

Telephone number of the local telephone.

`telephone_number`

Type: string
Defaults: -

Telephone number which is being called.

`connection_time`

Type: duration
Defaults: 0

The duration of the dial up connection.

direction

Type: string
Defaults: -

Direction of the connection. `Inbound` connections are those in which is machine on which the log is recorded is called, while in `outbound` connections the machine makes the call.

connection_type

Type: string
Defaults: -

The type of the connection: `speech` or `data`.

connect_status

Type: string
Defaults: -

The status of the connection: `busy`, `ring`, `failed` or `connected`. A failed connection happens when the dialed telephone number does not exist (is unallocated).

hangup_status

Type: string
Defaults: -

The status of the connection: `no_answer`, `unallocated` or `normal`. An unallocated telephone number is a number that does not exist.

cost

Type: number
Defaults: 0

The total cost of the dial up connection.

cost_currency

Type: string
Defaults: -

The currency in which the cost is expressed.

Schemas for the dns Superservice

DLF Schema for DNS service

Schema ID: `dns`
Timestamp Field: `time`

Each records in the DNS DLF schema represents a query that was made to the DNS server, zone transfers or other types of administrative information isn't represented in the schema.

Fields in the Schema

time

Type: timestamp
Defaults: 0

The time at which the query was processed by the server.

requesting_host

Type: hostname
Defaults: -

The host that made the request.

request

Type: hostname
Defaults: -

The content of the DNS request. DNS queries are usually about an hostname.

type

Type: string
Defaults: -

The record type that was requested. Common DNS record types are PTR, A, CNAME, etc.

resolver

Type: string
Defaults: -

This field contains `recurs` if the requests was recursive, that is probably made by a client for which we are configured as primary DNS server. Otherwise this field contains `norecurs` to denotes that the request wasn't recursive. Non-recursive requests are usually made by a DNS server which is processing a recursive request from one client.

Schemas for the dnszone Superservice

DLF Schema for DNS Zone service

Schema ID: `dnszone`
Timestamp Field: `time`

This DLF file is adequate to represent most common information about dnszone operations: approved/denied AXFR requests, completed zone transfers, loaded master and slave zones and denied dynamic DNS updates. See also the bind8_named2dlf manpage for some more info on the dnszone DLF format.

Fields in the Schema

`server`

Type: hostname
Defaults: -

Name of the DNS server.

`time`

Type: timestamp
Defaults: 0

The time of the event.

`axfr_host`

Type: ip
Defaults: -

IP address of the host requesting the AXFR.

`axfr_zone`

Type: hostname

Defaults: -

Name of the zone being requested. E.g. foo.example.com.

`axfr_what`

Type: string
Defaults: -

Either approved, denied or axfr.

`loaded_zone`

Type: hostname
Defaults: -

Name of the zone just loaded. E.g. foo.example.com.

`loaded_serial`

Type: number
Defaults: 0

Serial of the zone just loaded, as in the DNS SOA record. E.g. 2002071301 or 1024654055.

`loaded_role`

Type: string
Defaults: -

Either master or slave.

updatedenied_host

Type: ip
Defaults: -

IP address of the host requesting the update.

updatedenied_zone

Type: hostname
Defaults: -

Name of the zone being updated.

Schemas for the email Superservice

DLF Schema for Email service

Schema ID: `email`
Timestamp Field: `time`

The email DLF schema represents delivery done by an mail transfer agent between one *one* recipient. Each DLF record contains the information related to the process of delivering between one sender and one recipient. Each exchange between one sender and one recipient will results in *one and only one* DLF record. If one sender is sending to multiple recipients, the DLF file should contains one record for each recipient.

> **Warning**
>
> This DLF schema is different from the other ones because it contains pre-analysed data. It is currently in redesign and will become something that will more closely resemble what you can usually find in an email log file. Different information for each phase of the mail delivery process.
>
> That schema will probably become a derived schema from the new base schema.

Fields in the Schema

time

Type: timestamp
Defaults: 0

The time of the first event regarding that delivery. That should be the time the email enter the delivery process.

logrelay

Type: hostname
Defaults: -

The hostname on which the MTA is running.

> **Warning**
>
> That original intent of that field was to analyse email servers' farm which used syslog to centralize their logs. This method of analysis for multiple servers was never actually used and is considerd obsolete.

queueid

Type: string
Defaults: -

The queue identifier. This should be the identifier that can be used to reconstitute the delivery process from the native log file regarding the delivery that this DLF record represents.

msgid

Type: string
Defaults: -

The content of the Message-ID header of the delivered message.

from_user

Type: string
Defaults: -

The local part of the sender's email address.

from_domain

Type: hostname
Defaults: -

The hostname of the sender's email address.

from_relay_host

Type: hostname
Defaults: -

The hostname from which the email was received. For email submitted via the **sendmail** interface or other similar mechanism, this should be `localhost`.

from_relay_ip

Type: ip
Defaults: -

The ip address from which the email was received. For email submitted via the **sendmail** interface or other similar mechanism, this should be `127.0.0.1`.

size

Type: bytes
Defaults: 0

The size of the message.

`delay`

Type: duration
Defaults: 0

The time that it took to deliver the message. This should be equivalent to the time of the delivery event minus the value of the `time` field.

`xdelay`

Type: duration
Defaults: 0

> **Warning**
>
> This field was never actually used and is obsolete.

`to_user`

Type: string
Defaults: -

The local part of the recipient's email address.

`to_domain`

Type: hostname
Defaults: -

The hostname of the recipient's email address.

to_relay_host

Type: hostname
Defaults: -

The hostname to which the email was delivered. If the recipient is a local user and this was the "final" delivery, this should be `localhost`.

to_relay_ip

Type: ip
Defaults: -

The ip address to which the email was delivered. If the recipient is a local user and this was the "final" delivery, this should be `127.0.0.1`.

stat

Type: string
Defaults: -

The status code of the delivery. Only standardized code are `sent` which is used when the delivery succeeded without error and `deferred` for when the email still wasn’t delivered at the end of the log file. Other values are specific to each service.

xstat

Type: string
Defaults: -

The complete native status message related to the email’s delivery.

Extended Schemas for the email Superservice

Email Extended Schema for Email service

Schema ID: `email-email`
Base Schema: `email`
Module: `Lire::Extensions::Email::EmailSchema`
Required Fields: from_user, from_domain, to_user, to_domain

This is an extended schema for the Email service which adds fields containing the complete email.

Fields in the Schema

`from_email`

Type: email
Defaults: -

The sender's email address. That value is the *`from_user`* joined to the *`from_domain`* by an `@`.

`to_email`

Type: email
Defaults: -

The recipient's email address. That value is the *`to_user`* joined to the *`to_domain`* by an `@`.

Schemas for the firewall Superservice

DLF Schema for Firewall service

Schema ID: `firewall`
Timestamp Field: `time`

The firewall schema can be used for three types of logs: packet filtering firewall, intrusion detection system events and packet accounting devices.

Fields in the Schema

`time`

Type: timestamp
Defaults: 0

The time of the event.

`action`

Type: string
Defaults: -

What action was associated with that packet. Either `denied` or `permitted`.

`protocol`

Type: string
Defaults: -

The procotol of the packet. Common protocols are TCP, UDP or ICMP. This should be the IP protocol not higer-level application protocol.

`from_ip`

Type: ip
Defaults: -

The source ip address on the packet.

`from_port`

Type: port
Defaults: -

The source port (in the case of the TCP or UDP) protocol. This should be the ICMP type when the protocol is ICMP.

`from_host`

Type: hostname
Defaults: -

The hostname associated with the source IP.

`rcv_intf`

Type: string
Defaults: -

The receiving interface. That should be the network interface on which the packet was received. That field should contains the logical name or type of the interface.

`rcv_hwaddr`

Type: string
Defaults: -

The hardware address of the receiving interface. That's the MAC address in the case of an ethernet device.

`to_ip`

Type: ip
Defaults: -

The destination ip address on the packet.

to_port

Type: port
Defaults: -

The destination port (in the case of the TCP or UDP) protocol. This should be the ICMP code when the protocol is ICMP.

to_host

Type: hostname
Defaults: -

The hostname associated with the destination IP.

snt_intf

Type: string
Defaults: -

The sending interface. That should be the network interface on which the packet was sent (i.e. the outgoing interface).

length

Type: bytes
Defaults: 0

The packet length (that is the header and payload length). This should be the total length of the stream when the event represent multiple packets, for example, in the case of packet accounting done on streams.

rule

Type: string
Defaults: -

The rule that triggered that packet to be logged, denied, permitted, etc.

msg

Type: string
Defaults: -

A message associated with that packet. This could be an attack signature detected by a Network Intrusion Detection System or anything of similar nature.

count

Type: int
Defaults: 1

The number of packets described by this event. This will be 1 in the case of a single packet. It can be higher in the case where multiple packets are compressed into one event. Remember that the length values should reflect the length of all those packets.

Schemas for the ftp Superservice

DLF Schema for FTP service

Schema ID: `ftp`
Timestamp Field: `time`

This DLF file is adequate to represent most common informations about ftp transfers. It has the equivalent information of the xferlog format supported by many ftp servers.

Each DLF record in the FTP schema reprensents one FTP transfer, this schema isn't adequate to represents complete FTP session.

Fields in the Schema

`time`

Type: timestamp
Defaults: 0

The time at which the transfer started.

`transfer_time`

Type: duration
Defaults: 0

The time taken by the transfer.

`remote_host`

Type: hostname
Defaults: -

The hostname of the client that initiated the transfer.

`file_size`

Type: bytes
Defaults: -

The number bytes transferred.

`filename`

Type: filename
Defaults: -

The filename that was transferred.

`transfer_type`

Type: string
Defaults: -

The method used for the transfer. This will `ascii` when ASCII conversion is used during the transfer and `binary` otherwise.

`special_action_flag`

Type: string
Defaults: -

A code used to represent special action accomplished by the server during the transfer. Valid codes besides the default are `compress`, `uncompress` and `tar`.

`direction`

Type: string
Defaults: -

This should be `upload` for incoming transfer and `dowload` for outgoing transfer.

`access_mode`

Type: string
Defaults: -

The type of authentication used. Valid values are `anonymous` when the use used anonymous login; `guest` for guest login and `authenticated` for when the user was authenticated through valid credentials.

username

Type: string
Defaults: -

The name of the authenticated user. In the case of guest or anonymous logins, this should be the email address or other identifier given by the user on logon.

service_name

Type: string
Defaults: -

That's the name through which the service was invoked. This will usually be FTP.

auth_method

Type: string
Defaults: -

Additional authentication that was done on the connection. The only defined value besides the default is `ident` for the case when the connection was authenticated through RFC1931 authentication.

auth_user_id

Type: string
Defaults: -

The user identification related to the *auth_method* mode of authentication.

completion_status

Type: string
Defaults: -

This will be `complete` when the transfer completed successfully and `incomplete` when the transfer was aborted before it completes.

Schemas for the msgstore Superservice

DLF Schema for Message Store service

Schema ID: `msgstore`
Timestamp Field: `time`

Each DLF records in the msgstore schema represents one command on the server. Not all fields will be meaningful for every type of action.

Fields in the Schema

`time`

Type: timestamp
Defaults: 0

The timestamp of the event.

`localserver`

Type: hostname
Defaults: -

The hostname on which the message store is running. In the case of message store proxies, this will be the target server.

`client_name`

Type: hostname
Defaults: -

The host name of the client related to the event.

`client_ip`

Type: ip
Defaults: -

The ip address of the client related to the event.

`user`

Type: string
Defaults: -

The user name of the authenticated user associated with the connection.

`protocol`

Type: string
Defaults: -

The protocol used to access the message store. This will be usually one of `pop` or `imap`.

`prot_cmd`

Type: string
Defaults: -

The command executed by the client. Defined commands are `login`, `badlogin`, `select`, `create`, etc.

`session`

Type: string
Defaults: -

A session identifier used to relate the different records to one sesssion. Session identifier can be reused.

`messages_downloaded`

Type: int
Defaults: 0

Number of messages downloaded.

`bytes_downloaded`

Type: bytes
Defaults: -

Number of bytes downloaded messages.

`stored_messages`

Type: int
Defaults: 0

Number of messages stored on the server.

`stored_size`

Type: int
Defaults: 0

Size of the messages stored on the server.

`session_duration`

Type: duration
Defaults: 0

The length of the session.

> **Note:** FIXME: When this field should be defined. On all events or only on the quit event?

`status`

Type: string
Defaults: -

The status of the event. This includes the error messages or other success information (like an MMP redirection).

Schemas for the print Superservice

DLF Schema for Print service

Schema ID: `print`
Timestamp Field: `time`

Each record in a the print schema contains the information about one print job.

Fields in the Schema

`time`

Type: timestamp
Defaults: 0

The time at which the print job started printing.

duration

Type: timestamp
Defaults: 0

The time the print job took to print.

client_host

Type: hostname
Defaults: -

The client hostname (or ip address) that requested the print job.

user

Type: string
Defaults: -

The name of the user who requested the print job.

job-id

Type: string
Defaults: -

The identifier assigned to the job by the printing system. No uniqueness constraint are placed on this field. For example, it is possible for a printing system to reset the job identifiers after each restart of the system.

printer

Type: string
Defaults: -

The printer's name on which the job was printed.

num_copies

Type: int
Defaults: 1

The number of copies of the job that were printed.

num_pages

Type: int
Defaults: 0

The number of pages contained in the requested print job *not counting the copies*, i.e. the number of pages in one copies of the print job.

size

Type: bytes
Defaults: -

The size of requested print job.

billing

Type: string
Defaults: -

An identifier used to relate the job to an billing account.

Extended Schemas for the print Superservice

Sheet Count Extended Schema for Print service

Schema ID: `print-sheets`
Base Schema: `print`
Module: `Lire::Extensions::Print::PrintSchema`
Required Fields: num_pages

This is an extended schema for the Print service which adds a field that gives the number of physical sheets printed for a job.

Fields in the Schema

`num_sheets`

Type: int
Defaults: 1

The total number of pages printed in the print job. That is the *`num_copies`* times *`num_pages`*.

Schemas for the proxy Superservice

DLF Schema for Proxy superservice

Schema ID: `proxy`
Timestamp Field: `time`

This DLF file is adequate to represent most common informations about web proxy events. It has the same information as found in most proxy-like servers log files.

This schema is adequate for proxy servers beyond web proxys servers. It can be used for socks and other types of connection-level proxies.

The DLF schema was designed by studying the WebTrends Enhanced Log Format, squid log files and thinking about SOCKS type of server.

Fields in the Schema

time

Type: timestamp
Defaults: 0

The time at which the request was initiated.

client_ip

Type: ip
Defaults: -

The IP address of the client.

client_host

Type: hostname
Defaults: -

The hostname of the client.

user

Type: string
Defaults: -

If the client was authenticated, this field should contains the authenticated username.

duration

Type: duration
Defaults: 0

The time taken by the connection.

cache_result

Type: string
Defaults: -

Result code for the cache TCP_MISS, TCP_HIT, etc. List is available on Squid page, and in squid_access2dlf(1). All DLF converter should map their native value to the squid's one which is very complete and exhaustive.

req_result

Type: int
Defaults: -

HTTP result of the request. e.g. 200 or 404.

protocol

Type: string
Defaults: -

The protocol of the proxied request: ftp, http, https, telnet, etc.

transport

Type: string
Defaults: -

The protocol used between the client and the proxy server. This is probably TCP, but can be UDP in some case (like SOCKS or ICP).

dst_ip

Type: ip

Defaults: -

The ip address of the destination.

`dst_host`

Type: hostname
Defaults: -

The hostname of the destination. In the case of web proxy, that will be the website

`dst_port`

Type: port
Defaults: -

Port of the destination used in IP session

`operation`

Type: string
Defaults: -

This field should only be defined in the case of web proxy requests. This should contains the HTTP method requested like `GET` or `POST`.

`requested_url`

Type: url
Defaults: -

This field should only be defined in the case of web proxy request. It should contains the URL requested on the remote server.

bytes

Type: bytes
Defaults: -

The number of bytes transferred from proxy server to the client

type

Type: string
Defaults: -

This field should only be defined for web proxy servers, it should contains the MIME type of the HTTP request's result (e.g. text/html or image/jpeg).

rule

Type: string
Defaults: -

This field contains the configuration rule's name that was used to accept or deny to request.

useragent

Type: string
Defaults: -

The useragent used by the client. E.g. 'Mozilla/4.0 (compatible; MSIE 5.0; Win32)' or 'Outlook Express/5.0 (MSIE 5.0; Windows 98; DigExt)'

result_src_code

Type: string
Defaults: -

Code qualifying the next two fields. (i.e. NONE, DIRECT, PARENT_HIT, etc.) All DLF converter should map their native value to the squid's one which is very complete and exhaustive.

result_src_ip

Type: ip
Defaults: -

The IP address of the server which handled the request, i.e. destination or other cache

result_src_host

Type: hostname
Defaults: -

The hostname of the server that handled the request and gave the result.

result_src_port

Type: port
Defaults: -

Port on referring host used in IP session.

cat_action

Type: string
Defaults: -

This field contains either the value `block` or `pass`. It is used when access control is based on content filtering.

cat_site

Type: string
Defaults: -

If the proxy server is doing content analysis, this field should contains the category for the requested website.

catlevel_site

Type: int
Defaults: -

Level can be 1 or 2. 1 meaning "no no" categories. 2 meaning "family fun" categories. This was taken from the WELF specification.

cat_page

Type: string
Defaults: -

Like *cat_site*, but for the actual page.

catlevel_page

Type: int
Defaults: -

Like *catlevel_site*, but for the actual page.

Schemas for the syslog Superservice

DLF Schema for Syslog superservice

Schema ID: `syslog`
Timestamp Field: `timestamp`

This is a DLF schema that can be used to represent most messages logged through syslog-like daemon.

Fields in the Schema

`timestamp`

Type: timestamp
Defaults: 0

The timestamp of the logged event.

`hostname`

Type: hostname
Defaults: localhost

The hostname or ip address from which the message was received.

`process`

Type: string
Defaults: -

The "process" that logged the event.

`pid`

Type: int
Defaults: 0

The PID of the originating process that was included in the message.

`facility`

Type: string
Defaults: -

The syslog facility (`kern`, `mail`, `local7`, etc.) to which the message was logged. This information isn't present in all syslog file formats.

`level`

Type: string
Defaults: -

The syslog level (`emerg`, `notice`, `debug`, etc.) to which the message was logged. This information isn't present in all syslog file formats.

`message`

Type: string
Defaults: -

The logged event message (after the *`process`* and *`pid`* parts are removed).

Schemas for the www Superservice

DLF Schema for WWW service

Schema ID: `www`
Timestamp Field: `time`

In this DLF schema, each record represents a request to the web server. It has the equivalent information than the common log format supported by most web servers.

Fields in the Schema

`client_host`

Type: hostname
Defaults: -

The hostname (or ip address) of the clients that made the request.

`who`

Type: string
Defaults: -

If the request was authenticated, this field should contains the name of the authenticated user. Not that there is no indication of which authentication method was used (RFC1531, WWW authentication, etc.).

`http_result`

Type: string
Defaults: -

The numeric result code of the request. That's is `200`, `301`, etc.

`requested_page_size`

Type: bytes
Defaults: -

The number of bytes sent to the client during the request.

`http_action`

Type: string
Defaults: -

The method used by the client for the request. That is usually one of `GET`, `HEAD`, `POST`, etc.

`requested_page`

Type: url
Defaults: -

The URL that was requested by the client.

`http_protocol`

Type: string
Defaults: -

The protocol used by the client. It should usually be one of `HTTP/1.0` or `HTTP/1.1`.

`time`

Type: timestamp
Defaults: 0

The time of the request.

`referer`

Type: string
Defaults: -

The content of the `Referer` header that was sent along the request. That usually represents the referring URL, that's the URL which the user was browsing when this URL was requested.

useragent

Type: string
Defaults: -

The content of the `User-Agent` header that was sent along the request. That usually contains information the web browser used by the client.

gzip_result

Type: string
Defaults: -

When automatic compression is used, this should contains the result code from the compression submodule.

compression

Type: int
Defaults: 0

When automatic compression of the results is used, this field should contains the compression ratio achieved.

Extended Schemas for the www Superservice

Attack Extended Schema for WWW service

Schema ID: `www-attack`
Base Schema: `www`
Module: `Lire::Extensions::WWW::AttackSchema`
Required Fields: requested_page

This is an extended schema for the WWW service which tries to find common web attack based on the requested URL.

Fields in the Schema

attack

Type: string
Defaults: Unknown/No Attack

The type of attack that this request represents.

Domain Extended Schema for WWW service

Schema ID: `www-domain`
Base Schema: `www`
Module: `Lire::Extensions::WWW::DomainSchema`
Required Fields: client_host

This is an extended schema for the WWW service which adds a country and client_domain fields based on the client host.

Fields in the Schema

client_domain

Type: hostname
Defaults: -

The domain of the client host.

country

Type: string
Defaults: Unknown

The country of the client host as determined by the top-level domain.

Robot Extended Schema for WWW service

Schema ID: `www-robot`
Base Schema: `www`
Module: `Lire::Extensions::WWW::RobotSchema`
Required Fields: None

This is an extended schema for the WWW service which adds a robot field based on information from the domain name or the *`user_agent`* string.

Fields in the Schema

`robot`

Type: string
Defaults: Unknown/No Robot

The name of the robot that made the request.

Search Engine Extended Schema for WWW service

Schema ID: `www-search`
Base Schema: `www`
Module: `Lire::Extensions::WWW::SearchSchema`
Required Fields: referer

This is an extended schema for the WWW service which analyze the referrals. It extract the referring sites and it also determines if it was a request from a search engine.

Fields in the Schema

`referring_site`

Type: string
Defaults: -

The site which reffered that request. This is usually an hostname, but it can also be `bookmarks` for when the user used a bookmark.

search_engine

Type: string
Defaults: -

The name of the search engine, when the request was referred through a search engine.

keywords

Type: string
Defaults: -

The search phrase used when the request was referred through a search engine.

URL Extended Schema for WWW service

Schema ID: `www-url`
Base Schema: `www`
Module: `Lire::Extensions::WWW::URLSchema`
Required Fields: requested_page

This is an extended schema for the WWW service which parses the requested URL and adds several fields based on this information.

Fields in the Schema

requested_file

Type: filename
Defaults: -

The portion of the requested URL that represents a filename. That is everything that comes before the `?` which starts the `QUERY_STRING`.

requested_page_ext

Type: string
Defaults: -

The extension of the requested file.

directory

Type: filename
Defaults: -

The directory portion of the URL.

User Agent Extended Schema for WWW service

Schema ID: `www-user_agent`
Base Schema: `www`
Module: `Lire::Extensions::WWW::UserAgentSchema`
Required Fields: useragent

This is an extended schema for the WWW service which adds fields to access information from the *`user_agent`* field.

Fields in the Schema

browser

Type: string
Defaults: Unknown

The browser that was probably used to make the request as guessed from the *`user_agent`* field.

os

Type: string
Defaults: Unknown

The client's operating system as guessed from the *`user_agent`* field.

lang

Type: string
Defaults: Unknown

The client's language as guessed from the locale's information contained in the *user_agent* field.

Derived Schemas for the www Superservice

User Session Derived Schema for WWW service

Schema ID: `www-user_session`
Base Schema: `www`
Module: `Lire::Extensions::WWW::UserSessionSchema`
Required Fields: time, client_host
Timestamp Field: `session_start`

This is a derived schema for the WWW service which represents user session. User sessions tracks the traversal of users through the web site. Users are tracked using their IP address and their user agent information. This is not a full proof method. For one thing, it clearly fails in the case of users having an homogeneous environment and browsing from behing a proxy server.

Possible enhancements would be to use tracking information from a cookie.

The session represent all the consequential requests made by a user. The session will end after 30 minutes where no requests was made by the user.

Fields in the Schema

session_id

Type: string
Defaults: -

This field contains an arbitrary session identifier.

session_start

Type: timestamp
Defaults: 0

The time at which the session started.

session_end

Type: timestamp
Defaults: 0

The time of the last request in the session.

session_length

Type: duration
Defaults: 0

The length elapsed between the first and last requests.

page_count

Type: int
Defaults: 0

The number of pages requested by the user in this session. (This excludes requests ending in `.png`, `.jpg`, `.jpeg`, `.gif` and `.css`.)

req_count

Type: int
Defaults: 0

This gives the number of requests by the user

`first_page`

Type: filename
Defaults: -

The first page requested by the user. (See page_count for exlusion.)

`page_2`

Type: filename
Defaults: -

The 2nd page requested by the user.

`page_3`

Type: filename
Defaults: -

The 3rd page requested by the user.

`page_4`

Type: filename
Defaults: -

The 4th page requested by the user.

`page_5`

Type: filename
Defaults: -

The 5th page requested by the user.

last_page

Type: filename
Defaults: -

The last page requested by the user.

completed

Type: bool
Defaults: -

Was this session completed? A completed session is one that we know for sure that if the user made another request, it would have been in a new sesssion. Concretely, all requests made in the last 30 minutes of the period covered by the log file will be part of uncompleted sessions.

visit_number

Type: int
Defaults: 0

This starts at 1 for the first session of a user in the log file and will be incremented for each new session started by that user in the same log file.

IV. Lire Developers’ Conventions

Chapter 7. Contributing Code to Lire

The LogReport team invites you to contribute code to Lire. We're very happy with any code contributions which work for you: it'll very likely will make life easier for other people too! We ask you to consider some points, when writing code to get distributed with Lire.

When adding new scripts, or extending and improving current Lire code, make sure you're working with the current Lire code. (When working with old code, the bug you're working on might be fixed already by somebody else.) You can get the current code by fetching our CVS from SourceForge, using the anonymously accessible pserver:

```
cvs -d:pserver:anonymous@cvs.logreport.sourceforge.net:/cvsroot/logreport login
```

When prompted for a password for anonymous, simply press the Enter key.

```
cvs -z3 -d:pserver:anonymous@cvs.logreport.sourceforge.net:/cvsroot/logreport co service
```

See also the instructions on the SourceForge website (http://sourceforge.net/cvs/?group_id=5049). Alternatively, you can peek at the Lire CVS (http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/logreport/) using your webbrowser.

When you'd like to change e.g. `/usr/local/bin/lr_log2report`, you'll have to hack on `cvs/sourceforge/logreport/service/all/script/lr_log2report.in`. This file will get converted to `lr_log2report` by running `./configure`. Of course, when adding scripts or extending scripts, be sure to update the scripts' manpage too.

If you'd like the LogReport team to distribute your contribution, be sure to offer it to the team under a suitable software license. Refer to the Licensing section in the *Lire FAQ* for details.

Once you've tested your script, you can send it too the LogReport development list on development@logreport.org. The LogReport team will be happy to ship your contribution with the next Lire release.

Chapter 8. Developers' Toolbox

Required Tools To Build From CVS

In order to be able to build the program from the CVS tree and make a tarball distribution the following tools are needed:

- DocBook XML 4.1.2 (http://www.oasis-open.org/docbook/)
- DocBook DSSSL stylesheets (http://docbook.sourceforge.net/projects/dsssl/)
- autotools
- Jade (http://www.jclark.com/jade/) or OpenJade
- lynx (http://lynx.isc.org/)
- GNU make
- Perl's XML::Parser module
- dia
- epsffit
- epstopdf
- xsltproc
- xmllint

For Debian woody the packages are: docbook-utils (http://packages.debian.org/testing/text/docbook-utils.html), docbook-xml-stylesheets, autoconf (http://packages.debian.org/testing/devel/autoconf.html), automake (http://packages.debian.org/testing/devel/automake.html), autotools-dev (http://packages.debian.org/testing/devel/autotools-dev.html), jade (http://packages.debian.org/testing/text/jade.html), lynx (http://packages.debian.org/testing/web/lynx.html), make (http://packages.debian.org/testing/devel/make.html) and libxml-parser-perl.

Accessing Lire's CVS

Make sure you've got an account on *SourceForge* (http://www.sourceforge.net). Get yourself added to the logreport project. (Joost van Baal joostvb@logreport.org can do this for you.) Make sure your ssh public key is on the sourceforge server.

A full backup of the complete LogReport CVS as hosted on SourceForge is made weekly and written to `hibou:/data/backup/cvs/`.

CVS primer

If you have a Unix like system, make sure you have this

```
CVSROOT=:ext:cvs.logreport.sourceforge.net:/cvsroot/logreport
CVS_RSH=ssh
```

in your shell environment.

Of course, you could do something like

```
$ eval 'ssh-agent'
$ ssh-add
```

to get a nice ssh-agent running.

Now do something like

```
$ cd ~/cvs-sourceforge/logreport
$ cvs co service
```

There are also repositories called 'docs' and 'package'. In the former the webpages are located and in the latter the package files for Debian GNU/Linux and other distributions are kept.

Files can then be edited and commited:

```
$ vi somefile
$ cvs commit somefile
```

and get flamed ;)

Subscribe yourself to the commit list (commit-request@logreport.org), to get all commit messages, along with unified diffs.

SourceForge

Mailing Lists

Chapter 9. Coding Standards

Indentation should be four spaces. No tabs please.

See also Message-Id: <1028238571.1085.185.camel@Arendt.Contre.COM> on the development mailing list for some rationale on coding standards.

Shell Coding Standards

Shell scripts should run -e. Shell script should be portable. Refer to *http://doc.mdcc.cx/doc/autobook/html/autobook_208.html* (http://doc.mdcc.cx/doc/autobook/html/autobook_208.html).

Perl Coding Standards

Perl scripts should use strict, and run -w. Documentation should come in .pod format, documentation about script internals should be in perl comments.

No & in function call unless necessary.

Split long lines using hard return; try to respect the 72th column margin (this is kind of a soft limit).

Chapter 10. Commit Policy

Make sure your changes run on your own platform before committing. Try not to break things for other platforms though. Currently, Lire supported platforms are GNU/Linux (Debian GNU/Linux, Red Hat Linux, Mandrake Linux), FreeBSD, OpenBSD and Solaris.

Documentation should be updated ASAP, in case it's obsolete or incomplete by new commits.

CVS Branches

When doing major architectural changes to Lire, branches in CVS are created to make it possible to continue to fix bugs and to add small enhancements to the stable version while development continues on the unstable version. This applies mainly to the service repository. The doc and package repositories generally don't need branching.

BTW: A nice CVS tutorial is available in the Debian cvsbook package.

Hands-on example

A branching gets announced. Be sure to have all your pending changes commited before the branching occurs. After a branch has been made, one can do this:

```
$ cd ~/cvs-sourceforge/logreport
$ mv service service-HEAD
$ cvs co -r lire-20010924 service
$ mv service service-lire-20010924
```

or (with the same result)

```
$ mv service service-HEAD
$ cvs co -r lire-20010924 -d service-lire-20010924 service
```

Now, when working on stuff which should be shipped in the coming release, one should work in service-lire-20010924. When working on stuff which is rather fancy and experimental, and which needs a lot of work to get stabilized, one should work in service-HEAD.

Naming, what it looks like

Here is what branches schematically look like:

```
release-20010629_1 ---> lire-unstable-20010703 ---> HEAD
      \
       \
   lire-20010630 ---> lire-stable-20010701
```

In this diagram a branch named `lire-20010630` was created from the `release-20010629_1` tag. `lire-unstable-20010703` is another tag on the *trunk* (the *trunk* is the main branch). `HEAD` isn't a real tag, it always points to latest version on the trunk.

Creating a Branch

To create a branch, one runs the command **`cvs rtag -b -r release-tag branch-name module`**. Note that this command doesn't need a checkout version of the repository. For example, to create the `release-20010629_1-bugfixes` branch in the service repository, e.g. to backport bugfixes to version 20010629_1, one would use **`cvs rtag -b -r release-20010629_1 release-20010629_1-bugfixes service`**. When ready for release, this could get tagged as `release-20010629_2`.

The *`release-tag`* should exist before creating the branch. In case you want to branch from HEAD, use **`-r HEAD`**. E.g. **`cvs rtag -b -r HEAD release_1_1-branch service`**. Once Lire 1.1 gets released, tag it as `release_1_1`.

Accessing a Branch

To start working on a particular branch, you do **`cvs update -r branch-name`**. For example, to work on the `release_1_1-branch` branch, you do in your checked out version, **`cvs update -r release_1_1-branch`**. This will update your copy to the version `release_1_1-branch` and will commit all future changes on that branch.

Alternatively, you can also specify a branch when checking out a module using **`cvs co -r branch-name module`**. For example, you could checkout the stable version of Lire by using **`cvs co -r release_1_1-branch service`**.

To see if you are working on a particular branch, you can use the **`cvs status file`** command. For example, running **`cvs status NEWS`** could show:

```
===================================================================
File: NEWS              Status: Up-to-date

   Working revision: 1.74
   Repository revision: 1.74 /cvsroot/logreport/service/NEWS,v
   Sticky Tag: lire-stable
   Sticky Date: (none)
   Sticky Options: (none)
```

The branch is indicated by the `Sticky Tag:` keyword. If its value is `(none)` you are working on the `HEAD` branch.

To work on the `HEAD`, you remove the sticky tag by using the command **`cvs update -A`**.

Merging Branches on the Trunk

You can bring bug fixes and small enhancements that were made on a branch into the unstable version on the trunk by doing a merge. You do a merge by using the command **`cvs update -j` *`branch-to-merge`*** in your working directory of the trunk. Conflicts are resolved in the usual CVS way. For example, to merge the changes of the stable branch in the development branch, you would use **`cvs update -j lire-stable`**.

You should tag the branch after each successful merge so that future changes can be easily merged. For example, after merging, you do in a checked out copy of the `lire-stable` branch: **`cvs tag lire-stable-merged-20010715`**. In this way, one week later we can merge the week's changes of the stable branch into the unstable branch by doing **`cvs update -j lire-stable-merged-20010715 -j lire-stable`**.

Chapter 11. Testing

One week before release the software should be tested on all supported platforms. In between releases the system gets tested on various platforms on an ad hoc basis. When testing, use the to-be-released tarball. Run `make dist` to generate such a tarball. Releases are done about every month.

Chapter 12. Making a Release

Before making an official Lire release, it should have been tested on all supported platforms. A release shouldn’t be made unless Lire builds, installs and generates an ASCII report from all supported log files on all supported platforms. If this is not the case, the release should be delayed untill this is fixed.

Making a new release of Lire involves many steps:

1. Writing the final version number in NEWS.
2. Tagging the CVS tree.
3. Building the "Standard" Lire tarball.
4. Building the "Full" Lire tarball.
5. Building the Debian GNU/Linux package.
6. Building the RPM package.
7. Uploading the tarballs and making packages available.
8. Advertising the release.

Setting version in NEWS file

Inbetween releases, the NEWS file generally reads "version in cvs". This should of course be changed to e.g. "version 20011205".

Tagging the CVS

Run e.g. **`cvs tag release-20011017`**.

Building The "Standard" Tarball

The "Standard" tarball is the one that contains only the code needed to build and install Lire. It doesn’t contain required libraries like expat or XML::Parser. There is also a "Full" version of the tarball that includes those libraries.

1. Start from a fresh copy by running the command **`make maintainer-clean-recursive`** in the directory where you checked out Lire’s source code.
    a. Make sure that there are no tarballs in the `extras` subdirectory.
2. Set the version and prepare the source tree by running the command **`./bootstrap`**. (You can overwrite the pre-cooked version by doing e.g. **`echo ‘date +%Y%m%d‘-R-f-jvb-1 > VERSION`** . Make sure your version hasn’t got too many characters. Non-GNU tar chokes if pathnames in the archive are too long.)

3. Generate Makefiles

    a. Run **`./configure`**

4. Build Lire and create the tarball by running the command **`make distcheck`**.

    This will build a tarball `lire-`*`version`*`.tar.gz` and then make sure that the content of this tarball can be built and installed. If that command fails, Lire isn't ready to be released. Fix the errors before making the release.

5. Sign Lire's tarball with your public key. To do this with GnuPG, run **`gpg --detach-sign --armor lire-`*`version`*`.tar.gz`**.

    A file `lire-`*`version`*`.tar.gz.asc` will be created. Publish this file together with the tarball. Now, people downloading the tarball can verify its integrity by downloading the .asc as well as your public key, and running **`gpg --verify lire-`*`version`*`.tar.gz.asc`** .

Building The "Full" Tarball

The "Full" tarball is the one that contains the required Perl and XML libraries along with Lire's source code. This tarball should be called `lire-full-`*`version`*`.tar.gz`.

1. If you built the "Standard" tarball, you should move it someplace else along with its signature, because this procedure will overwrite it.
2. Start from a fresh copy by running the command **`make maintainer-clean-recursive`** in the directory where you checked out Lire's source code.
3. Add the tarballs of the required libraries in the `extras` subdirectory. These tarballs can be downloaded using **wget**.

    a. **`wget http://www.cpan.org/modules/by-module/XML/XML-Parser.2.30.tar.gz`**

    b. **`wget http://prdownloads.sourceforge.net/expat/expat-1.95.2.tar.gz`**

4. Set the version and prepare the source tree by running the command **`./bootstrap`**.
5. Build Lire.

    a. Run **`./configure`**

    b. Run **`make`**

6. Create the tarball by running the command **`make`** followed by the command **`make distcheck`**.

    This will build a tarball and then make sure that the content of this tarball can be built and installed. If that command fails, Lire isn't ready to be released. Fix the errors before making the release.

7. Rename the generated tarball to `lire-full-`*`version`*`.tar.gz`.
8. Sign Lire's tarball with your public key. To do this with GnuPG, run **`gpg --detach-sign --armor lire-full-`*`version`*`.tar.gz`**.

A file `lire-full-`*`version`*`.tar.gz.asc` will get created. Publish this file, together with the tarball. Now, people downloading the tarball can verify its integrity by downloading the .asc along with it, as well as your public key and running **`gpg --verify lire-full-`*`version`*`.tar.gz.asc`** .

Building The Debian Package

This is a raw unformatted dump of what we did to build and upload the Lire .deb.

```
              $ cd ~/cvs-sourceforge/logreport/package/debian
              $ vi changelog

:r !date --rfc

              $ cd /usr/local/src/debian/lire/debian/20010219
```

Run something like 'DIB_V=20020214 DIB_P=lire DIB_TARDIR=../archive/ ./debian-install-build'. This does:

```
              $ cd /usr/local/src/debian/lire/debian/20010219
              $ cp \
  ~/cvs-sourceforge/logreport/service/lire-20010219.tar.gz .

              $ tar zxf lire-20010219.tar.gz
              $ cd lire/20010418
              $ mv lire-20010418 lire-20010418.orig
              $ tar zxf lire-20010418.tar.gz
              $ cd lire-20010418
              $ mkdir debian
              $ cp \
   ~/cvs-sourceforge/logreport/package/debian/[^C]* debian/
```

Export the shell environment variable EMAIL, it should hold your email address, as it is to appear in the maintainers field of the package. (One could use 'dh_make --copyright gpl -s' on first time debianizing.) Build the .deb by running:

```
$ debuild 2>&1 | tee /tmp/build
```

Check the .deb:

```
$ debc | less
```

After having *really* tested it (dpkg -i, purge, etc.), optionally install it on any local apt-able websites you might have (Joost has one on http://mdcc.cx/debian/) and upload it to hibou's apt-able archive:

```
$ scp lire_20010418-1_all.deb \
```

```
 hibou.logreport.org:/var/www/logreport.org/pub/debian/dists/local/contrib/binary-all/admin

$ scp lire_20010418*.gz \
 hibou.logreport.org:/var/www/logreport.org/pub/debian/dists/local/contrib/source/admin/
$ scp lire_20010418*.*s* \
 hibou.logreport.org:/var/www/logreport.org/pub/debian/dists/local/contrib/source/admin/
```

Move the old debian stuff on hibou to hibou:/pub/archive/debian/ . Update the Packages file by running

```
$ cd /var/www/logreport.org/pub/debian
$ make
```

To upload it to the official debian mirrors:

```
vanbaal@gelfand:/usr...src/debian/lire/20010418% date; \
  dupload lire_20010418-1_i386.changes
Thu Apr 19 14:27:38 CEST 2001
Uploading (ftp) to ftp.uk.debian.org:debian/UploadQueue/
[ job lire_20010418-1_i386 from lire_20010418-1_i386.changes New dpkg-dev, announcement wil
 lire_20010418.orig.tar.gz, md5sum ok
 lire_20010418-1.diff.gz, md5sum ok
 lire_20010418-1_all.deb, md5sum ok
 lire_20010418-1.dsc, md5sum ok
 lire_20010418-1_i386.changes ok ]
Uploading (ftp) to uk (ftp.uk.debian.org)
 lire_20010418.orig.tar.gz 163.1 kB , ok (12 s, 13.59 kB/s)
 lire_20010418-1.diff.gz 32.6 kB , ok (3 s, 10.88 kB/s)
 lire_20010418-1_all.deb 222.4 kB , ok (16 s, 13.90 kB/s)
 lire_20010418-1.dsc 0.6 kB , ok (0 s, 0.60 kB/s)
 lire_20010418-1_i386.changes 1.2 kB , ok (1 s, 1.22 kB/s) ]
```

check ftp://ftp.uk.debian.org/debian/UploadQueue/

Building The RPM Package

Uploading The Release

To release a new distribution, publish the tarball on various places and send an announcement to the <announcement@logreport.org> mailinglist, stating the most interesting new features. Furthermore, add a newsitem to the news list of the website. We'll describe how to upload the tarball to various places.

The LogReport Webserver

Upload the tarball to the LogReport webserver like this:

```
$ scp lire-20001211.tar.gz hibou.logreport.org:/var/www/logreport.org/pub/
```

On hibou, do:

```
               $ cd /var/www/logreport.org/pub
$ chown .www lire-20010525.tar.gz
$ chmod g+w lire-20010525.tar.gz

$ tar zxf lire-20001211.tar.gz
$ rm current && ln -s lire-20001211 current
$ rm current.tar.gz && ln -s lire-20001211.tar.gz current.tar.gz
$ rm -rf lire-20001205
$ mv lire-20001205.tar.gz archive
```

Update the `README.txt` file: Run

```
$ cd /var/www/logreport.org/pub
$ ( echo \
  'current is the latest official release'; echo; ls -lF c* ) > README.txt
```

Check the symlink to the documentation stuff in the tarball.

Check if the stuff in http://logreport.org/pub/docs is still up to date.

Advertising The Release

SourceForge

In order to release a distribution on SourceForge (SF), you login with your SF account on the SF website. Once logged in you go to the project webpage (https://sourceforge.net/projects/logreport/) and choose *Admin*. Down at the bottom of that page is a a *[Edit/Add File Releases]* link (click it (https://sourceforge.net/project/admin/editpackages.php?group_id=5049)).

You are able to edit packages, like the Lire package in the LogReport project. To add a new release, choose *[Add Release]*. As a release name uses the date, like 20010407, assign it to the Lire package and then use the *Create This Release* button to makes it effective.

The next page shows 4 steps of which only one (step 2) is not straightforward. In that step you assign files to a release (.tar.gz, .deb, .rpm). These files should be uploaded to SF's Upload anonymous FTP site at ftp://upload.sourceforge.net/incoming/. Make sure the file is placed in the `/incoming` directory. Click *Refresh View* in Step 2 to add the files you uploaded to the FTP site. Check the files belonging to the

release and Click *Add Files*. In step 3, set Processor to any. Set file type to .deb and source.gz. Click update/refresh. Step 4: send notice. Done.

Freshmeat.net

On Freshmeat.net, releases are not released, but get announced only. These announcements attract a lot of attention. The webpage for the Lire package can be found at http://freshmeat.net/projects/lire/.

To announce a new release go to Lire - development branch (http://freshmeat.net/branches/14593/) webpage. Choose *Add Release* from the Project pull down menu in the light blue area. The rest is very straightforward.

Chapter 13. Website Maintenance

We give hints on how to upgrade the website: installing stuff from current CVS on http://logreport.org (http://logreport.org/).

Commits to the CVS tree of the website are automatically propagated to hibou. For more information on the markup language of the website, see the WJML documentation (http://logreport.org/doc/wjml/).

Documentation on the LogReport Website

Be sure the links to stuff under `/pub/current` are still alive. E.g. the files `TODO`, `dev-manual.html` and `user-manual.html` are linked to.

Publishing the DTD's

The DTD's are published as HTML on the website by using `hibou:/usr/local/src/dtdparse/dtdparse-2.0b2-LogReportPatched.tar.gz`, which is a patched version of Norman Walsh's dtdparse utility. Before the utility is run, make sure that the DocBook DTD is not included in the parsing process, because the DocBook DTD should not be published. This is done by changing the line:

```
<!ENTITY % load.docbookx     "INCLUDE"                              >
```

into:

```
<!ENTITY % load.docbookx     "IGNORE"                               >
```

The webpages are then generated with:

```
perl ~/dtdparse-2.0b2-patched/dtdparse.pl --title "XML Lire Report Markup Language" --outpu
perl ~/dtdparse-2.0b2-patched/dtdformat.pl --html lire.xml
```

The resulting `lire` directory can be tar-ed, gziped and unpacked again on hibou in the directory `/var/www/logreport.org/pub/docs/dtd/`.

The other two DTD's are HTML-ized similarly, but remember to change the title when running **dtdparse.pl**.

Chapter 14. Writing Documentation

Documentation which comes with the Lire tarball is maintained in four formats: plain text, Perl POD, DocBook XML and UML diagrams. We'll talk about all four of these here.

Plain Text

Small files like `README`, `NEWS`, `AUTHORS`, `doc/BUGS`, and `doc/TODO` are traditionally maintained in plain text format. We adhere to this common practice.

Perl's Plain Old Documentation: maintaining manpages

We use Perl's pod (plain old documentation) for manpages. Every file installed with Lire in `/usr/bin/` must have a manpage. Every file installed in `/usr/share/perl5/Lire/` and `/usr/lib/lire/` should have a manpage. It would be nice if the files in `/etc/lire/` were documented in manpages too. And perhaps for some files in `/usr/share/lire/xml/`, `/usr/share/lire/reports/`, `/usr/share/lire/filters/` and `/usr/share/lire/schemas/` manpages could be useful.

Since the files in `/usr/bin/` are commands, ran by Lire users, the manpages describing these should focus on the user perspective. Describing the inner workings and implementations of the commands is less important than describing why someone would want to run the specific command. If there's need to make some remarks on the internals of these scripts, a section called DEVELOPERS could be added to the manpage. The perl modules installed in `/usr/share/perl5/Lire/` and the commands in `/usr/lib/lire/` are not intended as interfaces for the user. Only people wanting to change or study the operation of Lire itself will interact with these files; therefore, the manpages should explain the inner workings and implementations of these files. The configuration files in `/etc/lire/` might be changed by users. These should be properly documented: in manpages or in the *Lire User's Manual*.

Docbook XML: Reference Books and Extensive User Manuals

The main documentation of the Lire project is done in DocBook XML 4.1.2. E.g. this document is maintained in DocBook XML, as is the *Lire User's Manual* and the *Lire FAQ*. The *Lire User's Manual* has more information about DocBook.

After editing the *Lire Developer's Manual* or the *Lire User's Manual*, you should run **`make check-xml`** to make sure the document is still a valid DocBook document. You should fix any errors before committing your changes.

If everything went right, documentation is built in txt, tex, html and pdf format by running **`make dist`**, or just **`make`** in `doc/`. We give some hints which might be helpful in case you have to build the documentation manually.

To generate PDF:

```
$ jade -t tex -d /path/to/DSSSL/docbook/print/docbook.dsl roadmap.xml
$ pdfjadetex roadmap.tex
```

The last step is actually done two or three times to resolve page numbers.

To generate HTML:

```
$ jade -t sgml -d html.dsl roadmap.xml
```

And now you can use the `html.dsl` in the `doc/source` directory. (If necessary, adjust it to reflect the location of your DSSSL stylesheets). Use lynx to generate TXT output from HTML with:

```
$ lynx -nolist -dump roadmap.html > roadmap.txt
```

UML Diagrams

The Unified Modelling Language (UML) is a set of definitions on how design diagrams are composed. These diagrams will help to document and understand the internals of Lire, and are used as such in this manual.

UML Editing

Several UML editors are available, but few are open source. Among these are Dia (general diagram editor for Gnome), ArgoUML (written in Java) and UML Modeler (http://uml.sf.net/) (UML specific editor for KDE). The latter was used to draw the diagrams found in CVS `/service/doc/uml-diagrams`.

Diagram Types

UML supports several diagram types. Two important ones are *class diagrams* and *sequence diagrams*. The former is used to depict the relations and associations between classes. Classes can be programs or modules. The latter is used to show how certain tasks are performed in time, and can be used to model the sequence of events.

V. Implementation Details

Chapter 15. Issues with Report Merging

In some cases, a merged report doesn't display the right information. We outline some worst case scenarios, and justify our implementation.

Suppose log file 1 ("requests" with "sizes") looks like:

request	size
A	12
B	11
C	10

while log file 2 looks like:

request	size
D	3
E	2
F	1

We report on the top 2 biggest requests, so the report from log 1 looks like:

request	size
A	12
B	11

while the report from log 2 would look like:

request	size
D	3
E	2

Now we change the superservice.cfg file to list the top-4 biggest items. A naive merge would lead to:

request	size
A	12
B	11
D	3
E	2

Of course, this should've been:

request	size

request	size
A	12
B	11
C	10
D	3

This effect does not occur when keeping the top-limit to the same value. However, when we're not reporting on distinct values in the log, but are summing, more horrible things might happen. Consider this: We want to report on the total size by client. Logs look like:

client	size
a	12
b	11
c	10

and

client	size
d	4
e	4
c	3

Reports from these logs would look like:

client	size
a	12
b	11

client	size
d	4
e	4

After naively merging, one would get:

client	size
a	12
b	11

In fact, the complete report should look like:

client	size

client	size
c	13
a	12

Luckily, the Lire merging algorithm is not *this* naive: in fact, the XML reports store a little more records than actually needed. This heuristic trick leads to sane merged reports in most cases. However, since this is merely a heuristic trick, it is no waterproof guarantee.

See the description of the guess_extra_entries routine in the Lire::AsciiDlf::Group manpage for more implementation details.

Chapter 16. Overview of Lire scripts

An overview of the main scripts involved. **lr_spoold** is the engine behind a Lire Online Responder. **lr_log2report** is the main Lire command line interface. The **lr_log2xml** command is a helper scripts. The **lr_xml2report** command can be used by the user to merge XML reports. The **lr_sql2report** is not yet fully integrated in the Lire system. The **lr_rawmail2mail** command manages a Lire client setup. The **lr_cron** is fired of by **cron**, in a cron-driven setup.

```
lr_spoold
|
\_ lr_check_service
\_ lr_spool
   |
   \_ lr_processmail
      \_ lr_getbody
      |
      \_ lr_log2mail
         |
         \_ lr_inflate
         \_ lr_log2xml
         |
         \_ lr_xml2mail
         |  \_ lr_xml2report
         |  \_ lr_mail
         |
         \_ lr_archive_log

lr_log2report
\_ lr_inflate
\_ lr_log2xml
\_ lr_xml2report

lr_log2xml
\_ <LR_SERVICE>2dlf
\_ lr_dlf2xml
\_ (lr_dlf2sql)

lr_xml2report
\_ lr_xml_merge
\_ lr_xml2<OUTPUTFORMAT>

lr_sql2report
\_ lr_sql2dlf
\_ lr_dlf2xml
\_ lr_xml2report

lr_rawmail2mail
\_ lr_getbody
\_ lr_deanonymize
\_ lr_xml2mail

lr_cron
```

```
\_ lr_log2mail
```

lr_spoold monitors a Maildir spool for each responder address. lr_processmail processes an email message with a compressed log file attached. Refer to the manpages for the gory details.

Chapter 17. Source Tree Layout

Service specific scripts should reside in $CVSROOT/service/<service>/script/. Configuration data should be in <service>/etc/. Service specific documentation in <service>/doc/.

Furthermore, in each subdirectory there should be a Makefile.am.

Glossary

Definitions of particular terms used in Lire.

DLF

See: Distilled Log Format

Distilled Log Format

Example 1. DNS DLF Excerpts

```
1010912574 10.0.0.2 121.68.134.195.in-addr.arpa PTR recurs
1010912574 10.0.0.2 121.68.134.195.in-addr.arpa PTR recurs
1010912592 10.0.0.2 120.67.123.212.in-addr.arpa PTR recurs
1010912600 10.0.0.2 207.7.178.212.in-addr.arpa PTR recurs
1010912600 10.0.0.2 tr16.kennisnet.nl A recurs
1010912616 10.0.0.2 120.67.123.212.in-addr.arpa PTR recurs
1010912630 10.0.0.2 207.7.178.212.rbl.maps.vix.com ANY recurs
1010912630 10.0.0.2 NLnet.nl ANY recurs
```

This is the generic log format used by Lire to normalise the log files from different products.

Currenlty, this normalised log is a simple ASCII format where each event is represented by one line. The information about the event is represented by fields separated by spaces. All non-printable ASCII characters are replaced by `?`. Spaces in a field's value are replaced by _ (an underscore). Each line must have the same number of fields. A DLF file doesn't contain any header information. Example 1 shows an excerpt of a DNS DLF file.

See Also: Superservice, DLF Schema.

DLF Schema

Information about the order of the fields in a DLF file, their types and what they represent is specified in the DLF's schema. Schemas are defined in XML files using the Lire DLF Schema Markup Language (LDSML). Lire's offers an API (only in Perl for now) to programmatically access the information of a schema.

Log files of many different products can share a common DLF schema that makes Lire's reports easily comparable.

Report

A report is what is generated by Lire. It consists of several subreports. Those subreports can be grouped into sections. The report is computed from the DLF file (and not the native log file) based on a configuration file which describes the subreports that make up the final report along with their parameters. (Consult the *Lire User's Manual* section *Customizing Lire* for more information.)

Service

Put simply, a service is a specific application that produces log files. It is usually the case that one application will be equivalent to one service. For example, the mysql service is used to process MySQL's log files.

But more precisely, a service is a specific log format. For example, the common service can be used for all web servers that support the Common Log Format. Similarly, the welf service can be used to process firewall log files written using WebTrends Enhanced Log Format.

In order to generate a report on it, the native log will be converted to the appropriate superservice's DLF schema

Subreport

A subreport is a particular view on the DLF log's data. Subreports are defined in XML files using the Lire Report Specification Markup Language (LRSML). (Although it defines subreports, it is called a Report Specification because a report is made up out of several subreports.) Example of a subreport would be *Requests by Hours of the Day*.

Subreports are defined for a particular DLF schema.

Superservice

A superservice is a collection of services that share the same DLF schema and report. It is used to group together applications (services) that offer the same kind of functionality.

Lire currently supports eight superservices: database, dns, email, firewall, ftp, print, proxy, and www.