

LTTng: The Linux Trace Toolkit Next Generation

A Comprehensive User's Guide (version 2.3 edition)

Daniel U. Thibault, DRDC Valcartier Research Centre

The contents are published as open-access documentation distributed under the terms of the Creative Commons Attribution 4.0 International Public License (CC-BY 4.0, <http://creativecommons.org/licenses/by/4.0>), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Defence Research and Development Canada

External Literature

DRDC-RDDC-2016-N036

June 2016

IMPORTANT INFORMATIVE STATEMENTS

Work conducted for project 15bl, HeLP-CCS: Host-Level Protection of Critical Computerized Systems

Template in use: SR Advanced (2010) Template_EN (2015-11-05).dotm

© Her Majesty the Queen in Right of Canada, as represented by the Minister of National Defence, 2016. *LTTng: The Linux Trace Toolkit Next Generation: A Comprehensive User's Guide (version 2.3 edition)* is made available under the Creative Commons Attribution 4.0 International Public License (CC-BY 4.0, <http://creativecommons.org/licenses/by/4.0>).

© Sa Majesté la Reine (en droit du Canada), telle que représentée par le ministre de la Défense nationale, 2016. *LTTng: The Linux Trace Toolkit Next Generation: A Comprehensive User's Guide (version 2.3 edition)* est rendu disponible sous la licence publique Creative Commons Attribution 4.0 International (CC-BY 4.0, <http://creativecommons.org/licenses/by/4.0/deed.fr>).

Abstract

The Linux Trace Toolkit Next Generation (LTTng) software tracer was thoroughly documented from a user's perspective. The behaviour of each command was tested for a wide range of input parameter values, installation scripts were written and tested, and a variety of demonstration use cases created and tested. A number of bugs were documented as a result (ranging from "simply broken" to "what exactly are we trying to do here?") which are being addressed as time passes and the project keeps advancing in capability, reliability, and ease-of-use.

Whereas the companion *LTTng Quick Start Guide* assumed the simplest installation and use scenario, the *LTTng Comprehensive User's Guide* delves into the nitty-gritty of using LTTng in all sorts of ways, leaving no option undocumented and giving detailed, step-by-step instructions on how to instrument anything (applications, libraries, kernel modules, even the kernel itself) for use with LTTng.

Significance to defence and security

The Linux Trace Toolkit Next Generation (LTTng) offers reliable, low-impact, high-resolution, system-wide, dynamically adaptable, near-real-time tracing of Linux systems. Being able to observe what goes on within the operating system and any applications running on it is an essential prerequisite for the cyber security of governmental and military computer systems of all scales. As the LTTng project has now reached a reasonable level of maturity and can be used in production contexts, it was felt necessary to create a document detailing in practical terms how to deploy, use, and extend it. Large parts of this document have already been used to create the LTTng on-line documentation (<http://lttng.org/docs/#>); this publication can only make LTTng more accessible to all interested parties.

Résumé

Le traceur logiciel Linux Trace Toolkit Next Generation (LTTng) a été documenté exhaustivement du point de vue de l'utilisateur. Le comportement de chaque commande a été testé pour une grande variété de valeurs de paramètres, des scripts d'installation ont été écrits et testés, et bon nombre de démonstrations utilitaires créées et testées. Une quantité de bogues ont été documentées par conséquent (allant de « c'est cassé » à « qu'est-ce qu'on essaie d'accomplir ici ? »), lesquelles succombent progressivement au passage du temps tandis que le projet continue de s'améliorer en capacité, fiabilité, et facilité d'utilisation.

Alors que le document complémentaire *LTTng Quick Start Guide* supposait le cas le plus simple d'installation et d'utilisation, le *LTTng Comprehensive User's Guide* plonge dans le détail de l'utilisation de LTTng de toutes sortes de façons, ne laissant aucune option sans documentation et donnant des instructions étape-par-étape et détaillées expliquant comment instrumenter n'importe quoi (applications, bibliothèques, modules du noyau, et le noyau lui-même) aux fins de LTTng.

Importance pour la défense et la sécurité

Le traceur Linux Trace Toolkit Next Generation (LTTng) est en mesure de tracer les systèmes Linux en temps presque réel de façon flexible (dynamiquement adaptable), universelle, détaillée, fiable et économique en coût de performance. Être capable d'observer ce qui se passe à l'intérieur du système d'exploitation et de ses applications est une condition *sine qua non* de la cyber sécurité des systèmes informatiques gouvernementaux et militaires de toute échelle. Comme le projet LTTng a atteint un niveau raisonnable de maturité et peut être utilisé dans des contextes de production, il nous a semblé nécessaire de créer un document détaillant en termes pratiques comment le déployer, l'utiliser et l'augmenter. Une large part de ce document a déjà été utilisée pour créer la documentation en ligne de LTTng (<http://lttng.org/docs/#>); cette publication ne peut que rendre LTTng encore plus accessible aux parties intéressées.

Table of contents

Abstract	i
Significance to defence and security	i
Résumé	ii
Importance pour la défense et la sécurité	ii
Table of contents	iii
List of figures	ix
List of tables	xi
Acknowledgements	xii
1 Introduction	1
1.1 The LTTng software tracer — Basic concepts	2
1.2 Scope of this document.....	3
1.3 Quick references	4
1.4 The content of the DVD-ROM	5
1.4.1 VirtualBox virtual machine	5
1.4.2 Software repository.....	6
1.5 How to read this document.....	6
1.6 Caveats	7
2 LTTng architecture, basic configuration and controls	9
2.1 LTTng components	9
2.1.1 The LTTng session daemon(s) (<code>lttng-sessiond</code>)	10
2.1.2 The LTTng consumer daemon(s) (<code>lttng-consumerd</code>).....	12
2.1.3 The LTTng command-line and custom control interfaces.....	12
2.1.4 The <code>liburcu</code> library.....	13
2.1.5 The <code>liblttng-ust</code> library.....	13
2.1.6 The <code>liblttng-ust-ctl</code> library	13
2.1.7 The LTTng kernel modules	13
2.1.8 The LTTng relay daemon (<code>lttng-relayd</code>).....	13
2.1.9 <code>babeltrace</code> and its library	14
2.1.10 LTTV	14
2.2 Tracing sessions and traces.....	14
2.2.1 Domains and the tracing group.....	16
2.3 Flow of control and data	19
2.3.1 Discard and overwrite modes	21

2.4	Types of probes that can be used with LTTng	24
2.4.1	Tracepoints	24
2.4.2	Perf	25
2.4.3	Kprobes.....	25
2.4.4	Kretprobes	25
2.4.5	Syscalls	26
2.5	Trace events	27
2.6	LTTng output files.....	28
2.6.1	The concept of “channel”	28
2.6.2	Output files and folders	31
2.7	Transient LTTng working files.....	36
3	Installation, testing and controls of LTTng	39
3.1	The Ubuntu LTTng suite	41
3.1.1	Using Ubuntu’s Software Centre.....	44
3.1.2	Using the Synaptic Package Manager	45
3.1.3	Using the command line	48
3.1.4	Adding the <code>lttngtop</code> and <code>lttv</code> packages to the Ubuntu packages.....	48
3.2	The DRDC (LTTng 2.3.0) suite	48
3.3	The LTTng source code suite	50
3.3.1	Fetching the LTTng tarballs	50
3.3.2	Preparing the system.....	54
3.3.3	Building and installing each LTTng tarball	54
3.3.4	Testing for the kernel modules and manually loading them.....	61
3.3.5	Registering the root session daemon with Upstart.....	62
3.4	Controlling the kernel and user-space session daemons.....	62
3.4.1	Comparison between session daemon control modes.....	66
4	Instrumentation of software in the user and kernel spaces.....	69
4.1	Instrumenting a C application (user-space)	69
4.1.1	Polymorphous tracepoints	70
4.1.2	Adding LTTng probes in software applications	71
4.1.3	Building instrumented software applications	81
4.2	Instrumenting a system function for just one application.....	94
4.3	Instrumenting a C++ application (user-space).....	94
4.4	Instrumenting a Java application	100
4.4.1	JNI and tracepoint providers.....	101
4.4.2	<code>liblttng-ust-java</code>	102
4.4.3	Another Java tracepoint provider example	103

4.5	Instrumenting a 32-bit application on a 64-bit system	107
4.5.1	userspace-rcu-32	107
4.5.2	lttng-ust-32	108
4.5.3	lttng-tools-32	108
4.5.4	lttng-tools-64 (/usr/src/lttng-tools)	109
4.5.5	Making a source of 32-bit user-space events	109
4.6	Instrumenting the kernel space	110
4.6.1	The kernel tracepoint header	111
4.6.2	The kernel tracepoint provider module	126
4.6.3	Installing the kernel tracepoint provider	127
4.6.4	Adding LTTng probes to the Linux kernel	128
4.6.5	Adding LTTng probes to a Linux kernel module	131
4.6.6	Adding LTTng probes to a custom kernel module	132
	References	141
	Annex A Configurations and notes	147
A.1	LTTng bash command completion	147
A.2	Making your own installation packages	147
A.2.1	Making a repository	148
A.2.2	Using the repository	149
A.3	Trace collisions	149
A.4	User groups	150
A.4.1	Group membership	150
A.4.2	Listing groups	151
A.4.3	Group creation	151
A.4.4	Joining and quitting a group	151
A.5	Namespaces	151
A.6	The ftrace facility	152
A.7	Updating the system call names	153
A.7.1	Use lttng-syscall-extractor	153
A.7.2	Generate the system call TRACE_EVENT() macros	155
A.8	Adding the LTTng PPA to the software sources	157
A.8.1	Getting the PPA's authentication key	159
A.8.2	Receiving a PGP key directly from a server	162
A.8.3	Downloading packages from the LTTng PPA	163
A.9	Java/JNI support	164
A.10	SystemTap support	165
A.11	The GNU gold bug	165
A.12	Setting compilation flags	165

A.13	The Linux dynamic loader.....	166
A.13.1	The library search path	166
A.13.2	The libraries sought	167
A.13.3	Initializations and finalizations.....	168
A.13.4	Making libraries findable.....	169
A.14	Kernel instrumented module templates and scripts	170
A.15	DRDC firewall palliation.....	179
A.15.1	Installing on a network-less system.....	180
A.15.2	Installing on a network-less virtual machine	180
Annex B	The <code>lttng</code> program	181
B.1	Synopsis.....	181
B.2	Description	181
B.3	Program options.....	182
B.4	Commands	183
B.4.1	<code>add-context</code>	184
B.4.2	<code>calibrate</code>	187
B.4.3	<code>create</code>	188
B.4.4	<code>destroy</code>	193
B.4.5	<code>disable-channel</code>	194
B.4.6	<code>disable-consumer</code>	195
B.4.7	<code>disable-event</code>	196
B.4.8	<code>enable-channel</code>	199
B.4.9	<code>enable-consumer</code>	205
B.4.10	<code>enable-event</code>	208
B.4.11	<code>list</code>	219
B.4.12	<code>set-session</code>	222
B.4.13	<code>snapshot</code>	223
B.4.14	<code>start</code>	229
B.4.15	<code>stop</code>	230
B.4.16	<code>version</code>	231
B.4.17	<code>view</code>	232
B.5	Exit values	234
B.6	Environment variables.....	235
B.6.1	<code>LTTNNG_SESSIOND_PATH</code>	235

Annex C The <code>lttng-sessiond</code> program.....	237
C.1 Synopsis.....	237
C.2 Description	237
C.3 Program options.....	237
C.4 Environment variables.....	240
C.4.1 LTTNG_CONSUMERD32_BIN.....	240
C.4.2 LTTNG_CONSUMERD64_BIN.....	240
C.4.3 LTTNG_CONSUMERD32_LIBDIR.....	241
C.4.4 LTTNG_CONSUMERD64_LIBDIR.....	241
C.4.5 LTTNG_DEBUG_NOCLONE.....	241
C.5 Limitations.....	241
Annex D The <code>lttng-relayd</code> program.....	243
D.1 Synopsis.....	243
D.2 Description	243
D.2.1 Example	243
D.3 Program options.....	244
D.4 Limitations.....	245
Annex E The <code>lttng-gen-tp</code> program.....	247
E.1 Synopsis.....	247
E.2 Description	247
E.3 Program options.....	248
E.4 Template file format	248
E.4.1 Example	248
E.5 Environment variables.....	249
E.5.1 CC	249
E.5.2 CFLAGS, CPPFLAGS, LDFLAGS	249
Annex F The <code>babeltrace</code> program	251
F.1 Synopsis.....	251
F.2 Description	251
F.2.1 Common Trace Format (CTF) definitions.....	251
F.2.2 babeltrace representation of CTF data	253
F.2.3 Examples	259
F.3 Program options.....	260
F.4 Environment variables.....	265
F.4.1 BABELTRACE_VERBOSE	265
F.4.2 BABELTRACE_DEBUG	265
Annex G The <code>babeltrace-log</code> program.....	267
G.1 Synopsis.....	267
G.2 Description	267
G.3 Program options.....	267

Annex H	The lttngtop program	269
H.1	Synopsis.....	269
H.2	Description	269
H.2.1	Key bindings.....	271
H.3	Program options.....	276
Annex I	The lttng-ust library.....	277
I.1	Description	277
I.2	Environment variables.....	277
I.2.1	LTTNG_UST_DEBUG	277
I.2.2	LTTNG_UST_REGISTER_TIMEOUT	277
Annex J	The lttng-ust-cyg-profile libraries.....	279
J.1	Synopsis.....	279
J.2	Description	279
	Fast profiling	279
	Verbose profiling.....	280
	Example.....	280
Annex K	The lttng-health-check function	283
K.1	Synopsis.....	283
K.2	Description	283
K.2.1	LTTNG_HEALTH_CMD.....	283
K.2.2	LTTNG_HEALTH_KERNEL.....	284
K.2.3	LTTNG_HEALTH_CONSUMER.....	284
K.2.4	LTTNG_HEALTH_APP_REG	284
K.2.5	LTTNG_HEALTH_APP_REG_DISPATCH.....	284
K.2.6	LTTNG_HEALTH_APP_MANAGE_NOTIFY	285
K.2.7	LTTNG_HEALTH_APP_MANAGE	285
K.2.8	LTTNG_HEALTH_HT_CLEANUP	285
K.2.9	LTTNG_HEALTH_ALL.....	286
K.3	Return value.....	286
K.4	Limitations.....	286
	List of symbols/abbreviations/acronyms/initialisms	287
	Glossary.....	291

List of figures

Figure 1: Overview of the LTTng software tracer.	2
Figure 2: Architecture of the LTTng software tracer.	9
Figure 3: Work flow of an LTTng tracing session.	15
Figure 4: User2 enjoys the benefits of belonging to the tracing group.	18
Figure 5: Flow of control and data during an LTTng trace.	20
Figure 6: A circular buffer and its individual sub-buffers' states.	21
Figure 7: A circular buffer in overwrite mode, showing the extra sub-buffer.	23
Figure 8: Channels and buffer multiplicity for the per-process-ID buffering scheme.	28
Figure 9: Channel structure of a tracing session.	30
Figure 10: Folder and file structure of a typical stored trace.	33
Figure 11: Folder and file structure of a trace using per-user-ID buffering.	34
Figure 12: Folder and file structure of a remotely stored trace (coming from hostname).	35
Figure 13: Overall LTTng work flow.	40
Figure 14: Dependencies and recommendations between the Ubuntu LTTng packages.	43
Figure 15: Functional break-down of the Ubuntu LTTng packages.	44
Figure 16: Ubuntu's Software Centre and LTTng.	45
Figure 17: Ubuntu's Synaptic Package Manager and LTTng.	46
Figure 18: Synaptic requests the package's dependencies and recommendations.	47
Figure 19: The Synaptic Package Manager is ready to apply the configuration changes.	47
Figure 20: The Synaptic Package Manager requests final approval.	48
Figure 21: The installation details of the <code>lttng-tools</code> package.	50
Figure 22: The <code>git.lttng.org</code> Web page, listing the available projects.	52
Figure 23: A project summary Web page.	52
Figure 24: A head branch page, listing that branch's commits.	53
Figure 25: A tag commit page.	53
Figure 26: Functional break-down of the LTTng packages.	55
Figure 27: Tracing using <code>sudo -H lttng</code> .	66
Figure 28: Tracing using <code>lttng</code> as a member of the tracing group.	67
Figure 29: Tracing using <code>lttng</code> or <code>sudo lttng</code> while not a member of the tracing group.	68
Figure 30: The tracepoint instrumentation work flow.	69

Figure 31: Building a statically linked instrumented application.....	81
Figure 32: Building an instrumented application using a static library.....	82
Figure 33: Building a dynamically linked, statically aware instrumented application.....	85
Figure 34: Building an instrumented application using dynamic loading.....	88
Figure 35: Relationships between source files and objects when instrumenting the kernel	114
Figure 36: Selecting the kernel options with <code>make menuconfig</code>	128
Figure 37: Selecting the software sources.....	157
Figure 38: The Other Software tab of the Software Sources.....	158
Figure 39: Adding a software source.....	158
Figure 40: The NO_PUBKEY error message.....	159
Figure 41: A PGP Public Key Server search results screen.....	160
Figure 42: The LTTng PPA's Web page.....	160
Figure 43: The LTTng PPA's technical details.....	161
Figure 44: Key search results.....	161
Figure 45: Key ID.....	161
Figure 46: The Software Sources Authentication dialog.....	162
Figure 47: The View Package Details link.....	163
Figure 48: The Package Details.....	163
Figure 49: A package's Details.....	164
Figure 50: The dependency tree of a basic tracepoint provider.....	168
Figure 51: The starting <code>lttngtop</code> window.....	270
Figure 52: The PerfTop display.....	271
Figure 53: The IOTop display.....	272
Figure 54: An IOTop display with two processes highlighted (in green).....	273
Figure 55: A process details display.....	274
Figure 56: Two preferences displays.....	275
Figure 57: An IOTop display sorted on bytes written (W).....	276

List of tables

Table 1: LTTng transient files	37
Table 2: LTTng versions held in the Ubuntu repositories	41
Table 3: Package and tarball equivalencies by source.....	51

Acknowledgements

Special thanks to Mario Couture and Frédéric Painchaud for encouragement and guidance.

Thanks to the LTTng team for all the help provided: Christian Babeux, Raphaël Beamonte, Yannick Brosseau, Julien Desfossez, Francis Deslauriers, Mathieu Desnoyers, Jérémie Galarneau, David Goulet, Matthew Khouzam, Simon Marchi, Alexandre Montplaisir, Philippe Proulx, Dominique Toupin...and many others. Thanks also to the people who made LTTng possible: Michel Dagenais, Karim Yaghmour and many others who worked on LTTng's predecessor, LTT: Werner Almesberger, Greg Banks, Andrea Cisternino, Tom Cox, Rocky Craig, Jean-Hughes Deschênes, Klaas Gadeyne, Jacques Gélinas, Philippe Gerum, Theresa Halloran, Andreas Heppel, Jörg Hermann, Stuart Hughes, Dalibor Kranjčič, Vamsi Krishna, Bao-Gang Liu, Andy Lowe, Corey Minyard, Paolo Montegazza, Takuzo O'Hara, Steve Papacharalambous, Jean-Sébastien Perrier, Frank Rowand, David Watson, Bob Wisniewski, and Tom Zanussi.

1 Introduction

Tracing (of any kind) can be used in a variety of contexts. When developing an application, one may frequently use a development environment’s facilities for stepping through code, examining variables dynamically along the way, to make sure the algorithm is doing what it should, or that the input received from other parts of the system (files, network data streams, system messages, etc.) is as expected, or to understand what goes on when a reproducible bug is triggered.

When stepping is not appropriate, tracing statements may be added to the application in order to create a log of its actions, decision points and state transitions, so that examination of the log may answer one’s questions. At this point one has reached the software engineering definition of *trace* which the Institute of Electrical and Electronics Engineers (IEEE) gives: “a record of the execution of a computer program, showing the sequence of instructions executed, the names and values of variables, or both” [1] [2] [3].

This definition can be broadened to become “a record of the execution of a computer system [...]”, where “computer system” is understood as any computing hardware unit: typically a computer workstation, possibly multi-core, likely connected to one or more networks. A number of CPU-level techniques allow the trace to record the execution of any application (regardless of source code availability), including the operating system itself, its device drivers, etc.

LTTng is the ultimate evolution of this concept. LTTng has extremely low overhead, allowing it to perturb as little as possible the traced system (this is important when analysing the behaviour of some component under heavy load circumstances, for example). It is fully re-entrant, so no special actions need be taken regarding signals, changing threads or interrupts (this is important when analysing the behaviour of interrupt-servicing modules, for instance). Its control variables and tracing data components are wait-free, so it will not cause deadlocks; in fact, a design decision was consciously made to make LTTng lossy in situations where the choice is between trace integrity and system perturbation (this is one of the reasons the overhead is so low). The traces are extremely high resolution: the event timestamps track individual processor cycles and events specify on which processor they occurred in multi-core systems.

Furthermore, LTTng is extremely agile: multiple traces can be running at once, started and stopped as needed, and most of their parameters can be changed on the fly. It will soon be possible, using LTTng, to analyse a trace as it is being captured, and to react to events or patterns of events by changing what is being traced dynamically. LTTng can also stream the trace produced by a system to a remote location for storage and analysis, using a minimal footprint daemon whose only task is to ship the data away over a network or serial connection of some sort.

LTTng does have a few limitations: it currently does not support kernel early boot tracing [4]. Integration with kernel crash dump tools is fairly recent and limited (see the `snapshot` command, Section B.4.13, under the *Core dump handler* heading).

This *Comprehensive User’s Guide* is a complement to the companion publication, the *LTTng Quick Start Guide*. The latter provides step-by-step guidance in the installation and initial use of the LTTng suite, explaining a few typical use cases and providing simple examples of the LTTng commands. The *Comprehensive User’s Guide* is much more thorough and explores a great number of caveats and alternate use cases.

1.1 The LTTng software tracer — Basic concepts

At the heart of LTTng is a tracing session manager, the session daemon. Following instructions it receives from a command-line client, it turns on and off the generation of tracing events from the kernel and user-space applications. The events deposit data in circular memory buffers, from which consumer daemons move the data to its ultimate storage.

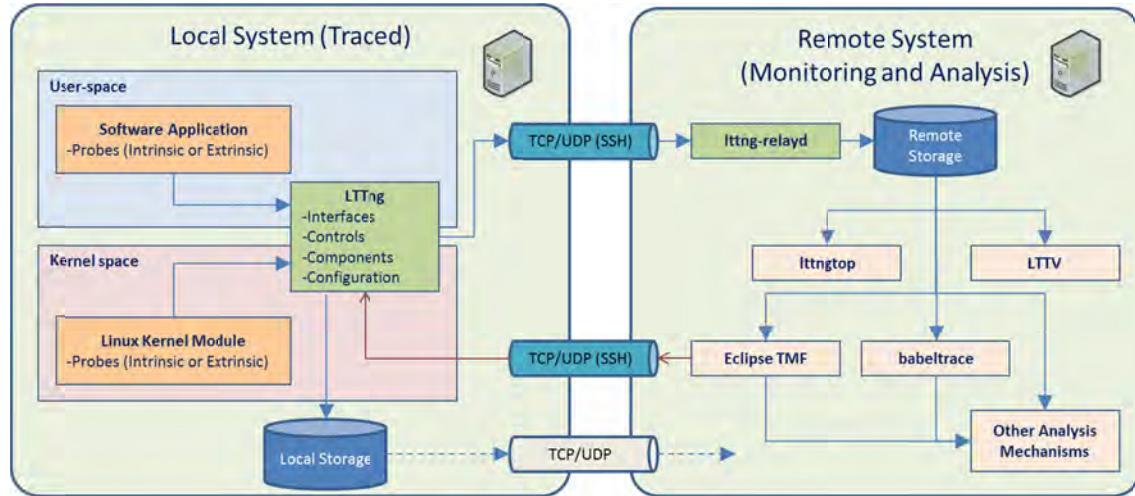


Figure 1: Overview of the LTTng software tracer.

Multiple tracing sessions may exist in parallel. Each user, for example, can run tracing sessions on applications running in his user-space. With super-user privileges, applications in any user-space can be traced, as well as the kernel itself. Multiple privileged users can share control of the “super-sessions”. Data from each session may be saved to local storage or streamed over the network for storage elsewhere.

The stored traces can accommodate a variety of tracing data formats, eventually allowing other tracing facilities to collaborate or cohabit with LTTng. A basic suite of tools is included to allow analysis of the traces to be conducted once a trace is completed. Currently, LTTng is in the process of finalising support for “live” trace analysis (appearing with version 2.4, released on March 2nd, 2014), so tools can safely work on stored trace data while more data is streaming in.

Probes can be intrinsic or extrinsic. Intrinsic probes are explicitly inserted in the source code, a process known as *instrumentation*. Extrinsic probes are injected into the memory image of executables running on the system, typically by making any access to chosen memory locations trigger a software interrupt of some sort. The two types of probes complete each other, allowing complete oversight of everything running on the system.

The main sort of tracing event source is the *tracepoint*, a type of intrinsic probe that can be dynamically activated and deactivated by the tracing session manager with very little impact on the instrumented objects. Tracepoints can readily be incorporated into C or C++ code; other languages, such as Java, are already supported and others may be in the near future. In addition to the kernel’s extrinsic `KProbe` mechanism, work is on-going to add a similar feature to user-space, allowing arbitrary executables to be instrumented.

Each tracing session organises its events of interest in *channels*, arbitrary groupings of events. This is done for two reasons. The first is *control*: from the user’s perspective (consider that the user may be a human operator but could also be a software system), the flow of events into each trace can be switched on and off at the event level, at the channel level, or globally, at the trace level. The second is *organisation*: each channel represents a single set of buffers, which in turn have a one-on-one correspondence with the trace’s logical files. Each channel can use a different buffer template (number and size of sub-buffers, part of the *buffering scheme*) in order to accommodate varying expected loads.

1.2 Scope of this document

This *Comprehensive User’s Guide* attempts to both document how LTTng currently works and how one goes about using it. It undoubtedly contains a number of errors and is certainly as yet incomplete. Because of this and also because LTTng itself is continuing to evolve, it should be seen as a first draft, and later editions should be expected. It is hoped it will serve not just as a how-to guide for tracing beginners, but also as a basis of discussion about LTTng’s future evolution for the designers and coders. The concepts of operation, for instance, have proven rather fluid, not just because they often were never written down other than in fragmentary fashion, but also because they have repeatedly evolved in response to the tracing community’s emerging needs.

The intended audience is thus rather broad. Beginners, including people with limited exposure to the Linux family of operating systems, should find it useful in accomplishing tasks ranging from the “simple” (installing LTTng and generating rudimentary traces) to the complex (adding experimental tracepoints to custom kernels) once the *Quick Start Guide* has reached its limitations. Experienced tracers should find it useful if only because it regroups under a single, searchable document the numerous `man` pages that concern LTTng’s various working parts. Finally, it is hoped the designers will find it useful in giving them a “forest perspective” (as opposed to a “tree perspective”) of the user experience and the overarching concepts of operation. At the very least, it would be appreciated if any errors are reported in order to be corrected in future editions of the document.

This document uses Linux Ubuntu 12.04.3 LTS as its platform of reference. LTTng runs on just about any Linux system: any mainstream Linux environment may serve for development, and LTTng can be used to trace running Linux systems on platforms with very limited on-board capabilities thanks to its remote trace streaming capability.

See Section 1.4 to find out about the Oracle VirtualBox virtual machine which is found on the DVD-ROM which accompanies this document. If you don't have a Ubuntu 12.04.3 machine handy, setting up the virtual machine is the fastest way to get going. Of course, the supplied virtual machine's basic configuration may need to be adapted to your circumstances (see Section 1.4.1).

For other Linux distributions (Fedora, Kubuntu, OpenSUSE, Red Hat, etc.), allowances will need to be made. Debian-based distributions should be able to make use of the software repository included on the DVD-ROM, whereas RPM-based ones will need to convert the packages or turn directly to the source code. Subtler changes may be required in exotic circumstances; for instance, the `lttng-ust` example makefiles need a small edit for BSD because the dynamic linker is not the same for that distribution.

The Eclipse LTTng Tracing and Monitoring Framework (TMF) was deliberately put out of scope. It has its own development community and deserves (and to some extent already has) extensive, separate documentation of its own.

1.3 Quick references

LTTng:

- ◆ LTTng Project Main Page: <http://lttng.org/>
- ◆ LTTng software repository (source code): <http://git.lttng.org/>
- ◆ EfficiOS software repository (babeltrace source code): <http://git.efficios.com/>
- ◆ LTTng PPA (Personal Package Archive): <http://launchpad.net/~lttng/+archive/ppa/>
- ◆ LTTng PPA software repository: <http://ppa.launchpad.net/lttng/ppa/ubuntu/>
- ◆ LTTng Bug Reports: <http://bugs.lttng.org/projects/>
- ◆ LTTng man pages: <http://lttng.org/files/doc/man-pages/>
- ◆ LTTng Tools Wiki (underdeveloped): <http://bugs.lttng.org/projects/lttng-tools/wiki/>
- ◆ Embedded Linux Wiki page on LTTng (April 2012, very short): <http://elinux.org/LTTng>
- ◆ Common Trace Format (CTF): <http://www.efficios.com/fr/ctf/>
- ◆ Distributed Multi-Core Tracing (Poly-Tracing) Research Project: <http://dmct.dorsal.polymtl.ca/>
- ◆ EfficiOS: <http://www.efficios.com/>

Ubuntu:

- ◆ Ubuntu Main Page: <http://www.ubuntu.com/>
- ◆ Ubuntu software repositories: <http://archive.ubuntu.com/ubuntu/>

Oracle VM VirtualBox:

- ◆ VirtualBox Main Page: <http://www.virtualbox.org/>
- ◆ VirtualBox software repository:
<http://download.virtualbox.org/virtualbox/debian/pool/contrib/v/virtualbox-4.3/>

OpenPGP key servers (SKS OpenPGP Public Key Servers):

- ◆ <http://keyserver.ubuntu.com/>
- ◆ <http://pgpkeys.mit.edu/>
- ◆ <http://pgp.mit.edu/>
- ◆ <http://pgp.openpkg.org/>
- ◆ <http://sks.pkgs.net/>
- ◆ <http://pool.sks-keyservers.net/>
- ◆ <http://zimmermann.mayfirst.org/>
- ◆ <http://http-keys.gnupg.net/>
- ◆ <http://sks.nimblesec.com/>

Mailing lists:

- ◆ LTTng Development Mailing List:
<http://lists.lttng.org/cgi-bin/mailman/listinfo/lttng-dev>

Miscellaneous:

- ◆ Oracle VirtualBox : <http://www.virtualbox.org/>

1.4 The content of the DVD-ROM

1.4.1 VirtualBox virtual machine

The DVD-ROM which accompanies this publication includes a `VirtualBox` folder in which can be found the Windows (bitness-independent) and Ubuntu 12.04 (32- and 64-bit) installers for Oracle VM VirtualBox 4.3.8, its multi-platform Extension Pack, and its SDK (Software Development Kit, also multi-platform) for good measure. The installer is also available for OS X, Solaris, and other Linux distributions (they can be obtained from https://www.virtualbox.org/wiki/Linux_Downloads). Up-to-date Ubuntu packages can be obtained by adding the VirtualBox PPA to Synaptic's list of software repositories (see Topical Note A.8):

```
deb http://download.virtualbox.org/virtualbox/debian precise  
contrib
```

The sub-folder `VirtualBox/VMs` contains a couple of virtual machines (as zipped archives that will need to be unzipped first). Both are fresh Ubuntu 12.04.4 LTS installations, one 32-bit and the other 64-bit, with a few updates detailed in the accompanying `README` file. The later snapshot adds LTTng 2.3 from the `drdc` software repository (see Section 1.4.2 below), with a few more tweaks, also detailed in the `README` file. The machines' configuration (the network card is a bridge to the host's, a network proxy had to be set up, the keyboard is French Canadian, and there is a shared folder set up with the host machine) will need to be adapted to your local configuration.

Either machine or snapshot should be enough to get you started right away. They each have two users set up: a main `user` account and a `tracer` account (member of the `tracing` group). Both users have `sudo` privileges.

1.4.2 Software repository

The DVD-ROM also includes a `drdc` folder which contains a software repository of the LTTng 2.3.0 (2013-Sep-06) suite. The Debian packages and scripts in the suite automate the compilation and installation. Using packages instead of compiling from a source tarball makes the various dependencies explicit, automates fetching support packages, and leaves an unmistakable audit trace of the installations in your package manager, letting you know exactly what is installed where as well as automating eventual uninstallations. The package scripts follow the procedures detailed in Section 3.3.3 as well as setting up the `tracing` group (see Section 2.2), registering the root `lttng-sessiond` daemon with Upstart (see Section 3.3.5) and setting up `lttng` command-line completion (see Topical Note A.1). An expanded set of examples was added to the `lttng-ust` package (see `/usr/src/lttng-ust-2.3.0/doc/examples/drdc` after installation is complete) and a few other minor adjustments made to the original git snapshots.

This document was prepared using this LTTng suite. To add the DVD-ROM's software repository to your system, see Topical Note A.2.2. If you opt for another installation procedure (see Chapter 3), some features of LTTng may turn out to be different from what is described here.

1.5 How to read this document

Chapter 2 explains LTTng's software architecture and discusses its major components. The terminology is defined and the process of generating traces detailed from an operator's perspective. Chapter 3 explains how to install LTTng starting from scratch. Chapter 4 uses a more programmer-oriented perspective, explaining how to instrument an application or how to add instrumentation to the kernel.

Annex A regroups topical notes, discussing various optional matters. Annex B through Annex H use a `man` page-like format to explain in elaborate detail what each command (`lttng`, `lttng-sessiond`, `lttng-relayd`, `lttng-gen-tp`, `babeltrace`, `babeltrace-log`, `lttngtop`), command parameter and command option does. The body of the document refers to these annexes nearly constantly, and they are meant as an important reference. Annex I through Annex K explain library (non-program) resources.

Refer to the Glossary, found near the end of the document, as necessary.

1.6 Caveats

- ◆ LTTng is still vigorously evolving, so this documentation will inexorably become more and more outdated. The on-line documentation (<http://lttng.org/docs/#>) was created mostly from this documentation and is expected to be kept up to date in the future.
- ◆ This document assumes a Ubuntu 12.04.3 LTS environment; procedures — particularly concerning installation— are likely to differ for other distributions. As users of other distributions supply their observations, they may be integrated into future editions.
- ◆ If you are used to the Eclipse environment, and have installed the Tracing and Monitoring Framework (TMF), you will be operating LTTng using a very different interface. This document’s observations concerning architecture, structure, and the consequences of configuration choices should nevertheless still stand.

This page intentionally left blank.

2 LTTng architecture, basic configuration and controls

2.1 LTTng components

Tracing with LTTng involves three active parts (the daemons), a number of supporting libraries, and a little bit of optional collaboration from the traced components. The traces thus generated can later be analysed using a number of tools. So-called live trace analysis —the analysis of traces which are still actively growing— is not yet possible but work is progressing rapidly in this direction. Figure 2, a more detailed version of Figure 1, will serve in the following discussion. This heading briefly tours the process of generating traces with LTTng; the sub-headings examine each component of LTTng in greater detail.

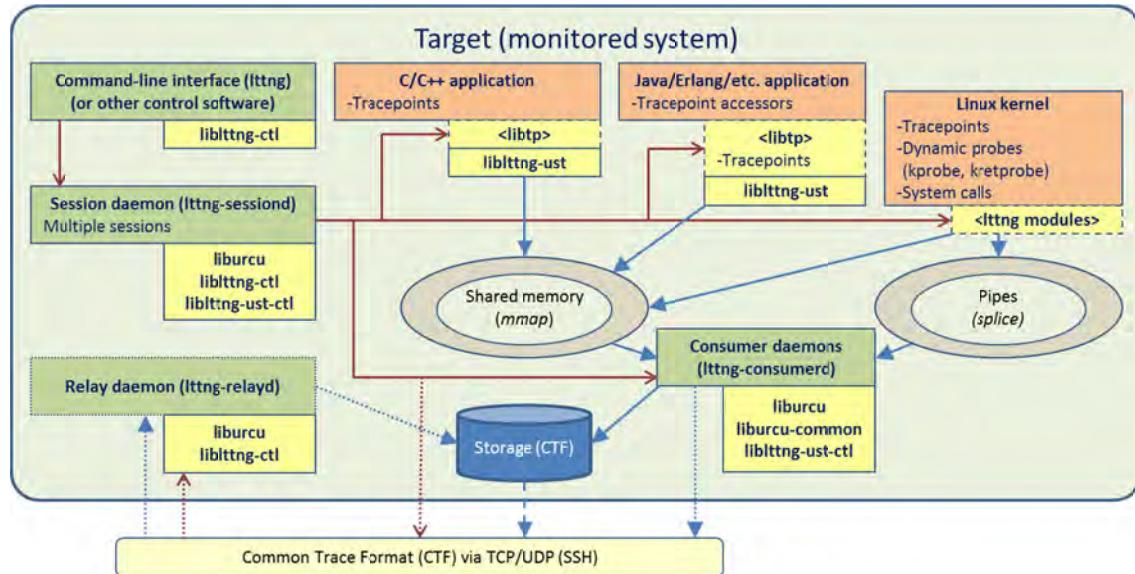


Figure 2: Architecture of the LTTng software tracer.

The LTng components are olive; instrumented processes are in light orange; libraries (direct dependencies) are in light yellow. Dashes indicate optionality. Control flow is in red, data flow in blue. Adapted from [5].

Commands are issued to the LTTng suite using a simple command-line interface (the `lttng` program, known as the *client*), or through direct exploitation of the LTTng application programming interface. The session daemon (`lttng-sessiond`) receives the commands and oversees all resulting tracing activity. The process of defining, starting, pausing, adjusting and concluding a trace is called a *tracing session*. Multiple tracing sessions may be active at any given time and are all managed simultaneously by the session daemon (the partitioning of sessions between the kernel and user-spaces is discussed later). The controlling role of the session daemon is shown by the control arrows emanating from its box in Figure 2.

Tracing events occur during a trace; consisting mostly of *tracepoints* although LTTng can accommodate a variety of other trace event generators such as kernel dynamic probes (`kprobes`). Tracing events are designed to be as unobtrusive as possible, so as to “steal” very few computing cycles and generally perturb normal operation of the traced system as little as possible. During each of their brief interventions, they write an *event record* to circular memory buffers set up by the session daemon. Each compact record notes which event it is, when it occurred, whence it came from (which CPU and process), and what its selected attributes are. The memory buffers are represented by the light brown rings in Figure 2, which also shows the data flow as blue arrows.

At this point one or more instances of another daemon, `lttng-consumerd`, intervene. Their job is to commit the event records to storage: it’s these daemons that actually write the trace. Storage can be local or it can be remote —that is to say, the event records can be sent over the network (dotted arrows in Figure 2). In that latter case, the third daemon (`lttng-relayd`) intervenes. It resides on the designated remote system (Figure 2’s box represents both the traced system and the remote storage system in this case) and its job is to receive the transmitted event records and commit them to storage. It is a simplified, specialised version of the `lttng-consumerd` daemon.

As event records are committed to storage, space is freed in the circular buffers for reuse by later event records. If the rate at which events are generated outstrips the rate at which they are consumed (committed to storage), the trace buffers will fill up, at which point LTTng will lose events. This also depends, to a lesser extent, on how the tracing buffers are set up (how many there are and how large they are). Which events to lose is part of the tracing session’s configuration: new events can either be discarded (early loss) or overwrite older events in the buffers (late loss).

Tracepoints, the main type of trace event, are implemented by small dynamic library fragments, the *tracepoint providers* (shown as “`(libtp)`” in Figure 2). These are shown as optional (dashed outlines) because, as Chapter 4 will show in detail, they are dynamically loaded: if the tracepoint providers are absent from the system, no events are generated and the instrumented components run with negligible overhead. In the case of the kernel, the tracepoint providers are kernel modules, a special type of shared object. Tracepoints are intrinsic, planned event sources, in the sense that they were inserted at the source code level. So is the kernel’s `syscall` instrumentation, which LTTng supports.

Other event sources, such as dynamic probes, are extrinsic (unplanned), in the sense that they are inserted forcibly into the executable object. This allows any application to be traced, regardless of source code availability. Mechanisms for dynamic instrumentation include trapping instructions (triggering exceptions when specific addresses are read or written; this is often used by debuggers) and inserting trampolines (replacing a small block of instructions with a jump to a handler which returns control to the original instructions at its conclusion). LTTng already supports the `kprobe` (and `kretprobe`) kernel instrumentation, and others are in the works.

2.1.1 The LTTng session daemon(s) (`lttng-sessiond`)

The `lttng-sessiond` daemons handle all tracing sessions on a per-user basis.

Multiple `lttng-sessiond` instances may exist, each belonging to a different user and each operating independently of the others. Only the root `lttng-sessiond` daemon may control the kernel tracer, however. There may be only one `lttng-sessiond` instance per user, including root (i.e., the kernel). Tracing sessions are local to each daemon; the scope of each event source depends on its domain and is explained in Section 2.2.1.

The LTTng command-line interface (and the LTTng API as well) deals exclusively with the session daemon; the latter in turn instructs the tracers and the consumer and relay daemons as needed. Later sections of this document explain each of the numerous tracing session configuration options.

One of the things which `lttng-sessiond` does is to keep track of the available event types. Each user-space tracepoint provider actively registers itself with `lttng-sessiond` upon being loaded into memory¹; by contrast, `lttng-sessiond` actively seeks out and loads the LTTng kernel modules (see Section 2.1.7) as part of its own initialisation. Kernel event types are *pulled* by `lttng-sessiond`, user-space event types are *pushed* to it by the user-space tracepoint providers.

2.1.1.1 The root session daemon as a service

When installing from an `lttng-tools` package, the root `lttng-sessiond` daemon will be registered as an Upstart service (see Section 3.3.5) and is launched as soon as the installation is completed. When installing from a tarball (see Section 3.3.3.4), service registration does not occur but you can easily do this manually (see Section 3.3.5).

Whatever your configuration may be, the `lttng-sessiond` daemon will start as soon as you issue commands that require its presence (some very simple commands, such as ‘`lttng`’ or ‘`lttng --help`’, do not trigger the daemon).

If the root `lttng-sessiond` daemon is managed by Ubuntu’s Upstart, it should not be killed (should the need arise —an unlikely scenario) by the `sudo kill` command because if you do so `lttng-sessiond` is immediately respawned by Upstart. Rather, you use the `service` command to stop or (re)start the daemon:

```
$ sudo service lttng-sessiond start  
$ sudo service lttng-sessiond stop  
$ sudo service lttng-sessiond restart
```

For instance, once you’ve installed the `lttng-tools` package from the accompanying DVD-ROM’s software repository, you can make sure the daemon is running by issuing the `restart` command given above.

¹ More precisely, the tracepoint provider registers itself with the root `lttng-sessiond` daemon, then with the local `lttng-sessiond` daemon. See `lttng-ust/liblttng-ust/lttng-ust-comm.c`.

2.1.2 The LTTng consumer daemon(s) (`lttng-consumerd`)

These daemons are created as soon as events are enabled within a session (see the `lttng enable-event` command, Section B.4.10), well before the session is activated by the `lttng start` command. Managed by the session daemon, the consumer daemons survive session destruction, to be re-used if a new session is created later. Consumer daemons are always owned by the same user as their session daemon, and when the `lttng-sessiond` daemon is stopped, the `lttng-consumerd` daemons also stop. This is because they are children of the `lttng-sessiond` daemon.

There are up to two consumer daemons per user, handling that user-space’s events. This is because each process has independent bitness: if your system runs a mixture of 32- and 64-bit processes, it is more efficient to have separate 32- and 64-bit consumer daemons. Root may have up to three consumer daemons, the third daemon handling the kernel events. The kernel won’t run simultaneously in two bitness modes, so the kernel events have only one bitness.

The C language specifies 64-bit wide types such as `long longs` and `long doubles` regardless of the bitness of the compiled application, so both the 32- and 64-bit consumer daemons have the same event record handling capabilities. It is theoretically possible to have a 64-bit consumer daemon service both 32- and 64-bit ring buffers (the ring buffer storing a process’s trace event records is itself in a context of the same bitness as the process), but having separate 32- and 64-bit consumer daemons and LTTng libraries simplifies compilation and linking [6].

As new domains are added to LTTng, the development community’s intent is to minimise the need for additional consumer daemon instances dedicated to these new domains. For instance, with the LTTng 2.4 `jul` domain, the `java.util.logging` events are mapped to user-space, so tracing this new domain is handled by the pre-existing user-space consumer daemons.

2.1.3 The LTTng command-line and custom control interfaces

The `lttng` client command-line program (see Annex B) is the user’s main interface to the LTTng suite. It acts as a “wrapper” around the `lttng-ctl` library, conducting command-line translation, some argument validation, and error reporting.

2.1.3.1 The `liblttng-ctl` library

LTTng’s `liblttng-ctl` is the control API used by the `lttng` program and other software, such as Eclipse’s Tracing and Monitoring Framework plugins, to control `lttng-sessiond`. The library manages the conversation with the session daemon, packaging the commands sent to it and unpackaging the responses returned.

2.1.4 The liburcu library

Despite its name (“user-space read-copy-update”), this library is also used during kernel tracing (`lttng-tools` needs `liburcu`—supplied by the `userspace-rcu` package— even when no user-space support is provided). It implements a user-space version of the read-copy-update (RCU) synchronisation mechanism widely used in the kernel since its introduction in 2002. RCU allows control parameters to be updated in read-side lock-less fashion [7]. Simply put, whenever a control parameter (data structure) needs to be updated, it is copied, the copy is modified, and the pointer to the new parameter is atomically swapped for the original. Shortly thereafter, a read-side quiescent state is reached where all the processes using the old pointer have completed their execution, and the old parameter copy can be reclaimed. Thus, during a short grace period there are both “old” processes that are reading the old parameter and “new” processes that are reading the new value. After this grace period the old parameter can be safely garbage-collected.

2.1.5 The liblttng-ust library

This library supplies user-space tracing (UST) support to user-space tracepoint providers.

2.1.6 The liblttng-ust-ctl library

This library supplies the LTTng control API parts specific to user-space tracing.

2.1.7 The LTTng kernel modules

The LTTng kernel modules, located in `/lib/modules/<kernel_version_name>/extra`, act as the kernel’s tracepoint providers (you can get the `<kernel_version_name>` from the `uname -r` command). The modules are available as soon as they are installed; they are, however, only loaded when needed, which is to say when a root `lttng-sessiond` daemon is running (see also Section 3.3.4). The user-space daemons don’t load the kernel modules because they do not interact with them at all.

2.1.8 The LTTng relay daemon (`lttng-relayd`)

This daemon is a simplified consumer daemon. It listens on the network for instructions from remote session daemons and data from remote consumer daemons. It operates independently from any local session and consumer daemons. If your operational configuration calls for most tracing sessions to send their data to a certain network host, it may be useful to register the relay daemon as a service like the root session daemon (see Section 3.3.5).

There are currently no discovery or remote startup facilities in LTTng: you must start the relay daemon from its host using other means, such as an SSH connection. Eclipse’s Tracing and Monitoring Framework (TMF) includes some discovery and remote startup facilities.

Whereas the consumer daemon (see Section 2.1.2) may exist in two bitnesses, the relay daemon need only be of its host operating system’s bitness. The data flows it receives are conveyed by the bitness-independent TCP (Transmission Control Protocol). Future plans include allowing UDP (User Datagram Protocol) as well, which is also bitness-independent.

The relay daemon is pivotal to LTTng’s “live” tracing facility (starting with version 2.4, released on March 2nd, 2014): live traces must be configured to send their data streams to “remote” destinations, which could happen to be the local workstation (the data stream then goes through the local loopback and is received by the local relay daemon) [8].

2.1.9 babeltrace and its library

A key enabler, the Common Trace Format (CTF), was designed in order to be as flexible as possible, satisfying the requirements of the embedded, telecommunications, high-performance, and kernel communities. The CTF specification can be obtained from EfficiOS at <http://www.efficios.com/fr/ctf>. Naturally, not all features of CTF are used by LTTng traces.

Babeltrace is a trace viewer and converter reading and writing the Common Trace Format (CTF). Its main use is to pretty-print CTF traces into a human-readable text output sorted in strictly increasing time. Babeltrace is expandable in plugin fashion in order to support reading and writing other trace formats. Its API, encapsulated in `libbabeltrace`, can be used by other applications (`lttv`, `lttngtop`, Eclipse TMF, etc.) to read CTF traces.

2.1.10 LTTV

LTTV is the Linux Trace Toolkit Viewer; as an interactive graphical visualization application, it allows sophisticated trace analysis comparable to the Eclipse LTTng visualization plugins. LTTV also has a command-line mode which can be used to conduct batch processing of traces, such as statistical calculations.

A complete description of this graphical tool is beyond the scope of this document. See `/usr/src/lttv/doc/user/user_guide/html/index.html` for the *Linux Trace Toolkit Viewer User Guide*.

2.2 Tracing sessions and traces

The *tracing session* is the fundamental LTTng object: it consists of a dynamic set of tracing parameters which defines tracing *event types*, groups event types in *channels* within *domains*, specifies where the captured data are to be stored, and controls tracing activity. The notion of domain is explained in detail in section 2.2.1 below. Events and channels are explained in detail in Sections 2.5 and 2.6.1 respectively; this heading gives just a brief description.

An *event* occurs when execution of a given process reaches a certain point. Event occurrence generates an *event record* in one or more traces (defined below). Each event record has a number of attributes such as the event’s type, its provenance and time of occurrence, and a “payload” of values of interest, which depends strongly on the type of event.

The channel is a construct peculiar to LTTng. It is a means of arbitrarily grouping event types in order to more easily control the flow of their occurrences into the trace (by flicking a channel on and off, one turns the flow of the channel’s events on and off) and also in order to organise them in logical groupings for later analysis.

Tracing sessions are created by the `create` command (see Section B.4.3) and destroyed by the `destroy` command (see Section B.4.4). Tracing session names are nearly arbitrary, although they are some file system-related restrictions. In particular, the sequence `/.../` must not appear in the session name (nor can the name begin with `../`), as LTTng won't be able to create the trace directory, be it local or remote [9] (starting with version 2.4, session names cannot contain any `/` characters; this fix was back-ported to `git.lttng.org`'s version `stable-2.3` because of the severity of the security issue this represented).

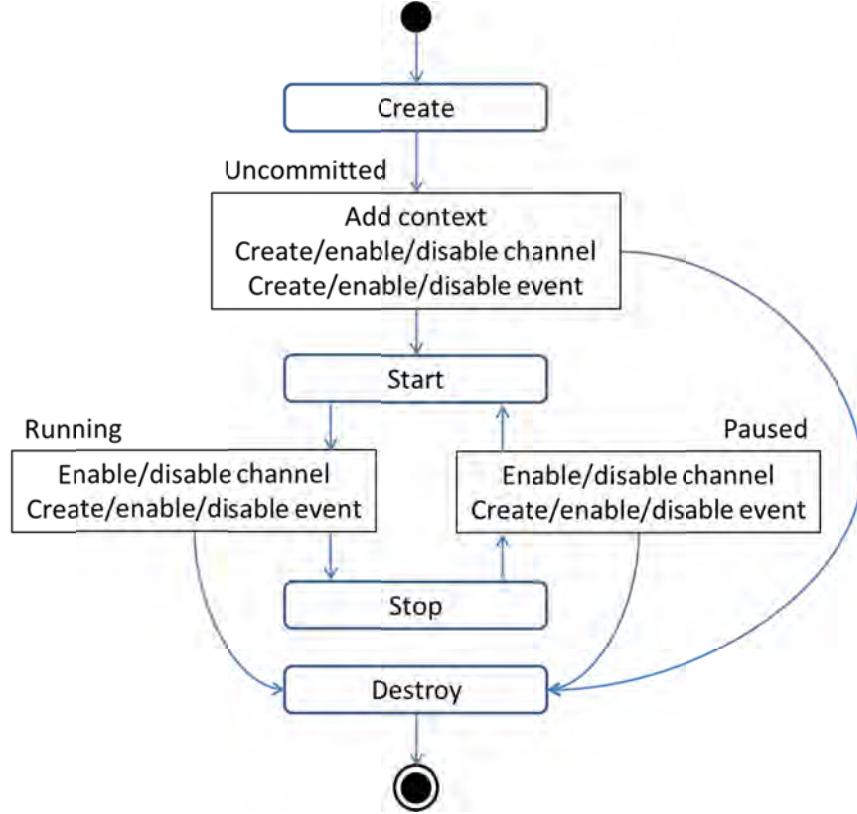


Figure 3: Work flow of an LTTng tracing session.

Figure 3 illustrates the life cycle and work flow of an LTTng tracing session. The `create` command sets the session's output options: local or remote storage, or no output at all (snapshot-taking mode: see the `snapshot` option of the `create` command, Section B.4.3). The tracing session is in an “uncommitted” state at this point. You can add context to channels, create channels (setting the domain buffering scheme with each domain's first channel) and create events. Once the configuration is complete, the tracing session can be started to become “active” or “running”. The `stop` command puts the session in the “paused” state (`lttng` reports both the “uncommitted” and “paused” states as “inactive”). Channels and events can be enabled and disabled while the tracing session is running or paused; new events can be created too, but not new channels. The `destroy` command can be used at any point to terminate the session.

Enabling and disabling channels and events works in conjunction: an event and its channel must both be enabled in order for the event to be captured. Enabling and disabling channels gives gross control, affecting whole groups of events at once, while enabling and disabling events affords fine control. LTTng affords even finer control through its event filter mechanism (see the `filter` option of the `enable-event` command, Section B.4.10).

Each user is free to manage multiple sessions at once: all session-specific commands accept a `session` option (see Section B.4), and the user’s environment keeps track of the freely-settable “current session” for convenience (see the `set-session` command, Section B.4.12).

Although sessions share the session and consumer daemons, they do not impinge on each other in any way. Sessions are said to be *active* (“running”) or *inactive* (“uncommitted” or “paused”) depending on whether or not trace events are being captured. The `start` (see Section B.4.14) and `stop` (see Section B.4.15) commands switch a session between the two states.

As explained in Section 2.1.1, there are multiple `lttng-sessiond` daemons. Each tracing session exists within (is handled by) one of those daemons. Session names have daemon scope, meaning it is possible to have two completely different sessions bearing the same name as long as they belong to different session daemons (e.g. a root session daemon and a local user session daemon). Furthermore, a session name may be re-used under a given daemon without fear of compromising the trace data, because the trace paths are constructed from the session names and their creation timestamps (see Section 2.6.2). The only restriction is that the homonymous sessions cannot exist simultaneously: they must exist sequentially, the first instance being destroyed before the second can be created. See Topical Note A.3 for a discussion of possible trace name collisions.

The *trace* is the set of files containing the data generated by the session (mostly event records with their timestamps, identifiers and attributes). Destruction of a session (see the `destroy` command, Section B.4.4), despite the ominousness of the operation’s name, does not delete the trace—it shuts down and deletes the *session*, so it is no longer handled by `lttng` itself. When a tracing session is finally destroyed, its data files (the trace) are closed and can be safely manipulated (copied, archived, etc.). The trace files of a completed tracing session are referred to as an “offline trace”. The `babeltrace`, `lttv` and `lttngtop` trace viewers can only handle offline traces for now, although live versions are in preparation.

2.2.1 Domains and the tracing group

Like most operating systems, Linux distinguishes between “kernel space” and “user space”. In LTTng parlance, these are two distinct *domains*² (it is conceivable that other domains may be defined in the future). Processes in the kernel domain are privileged with respect to memory and interruptibility when compared with those in user-space. In particular, user-space memory is fully virtual (i.e. normally swappable), whereas kernel memory is non-swappable.

² This should not be confused with the much more common usage of “domain” to designate collections of networking addresses or of networking and computing resources.

The concept of *hierarchical protection domains*, also called *protection rings*, was introduced in the late 1960s by the Multics operating system, an ancestor of Unix and Linux. Access to hardware resources increases as one moves from the outer ring towards the innermost ring (ring 0), and the gateways between rings are carefully monitored. Protection rings are enforced at the hardware level by the CPU’s operating modes (typically four nowadays, although most operating systems use only two rings, the innermost and outermost). In Linux, kernel operations are all performed within ring 0, whereas user-space resides in ring 3. One important consequence of this architecture is that transitions between different CPU operating modes are costly—hence LTTng goes to great lengths to minimise such transitions during tracing.

Because the separation between the kernel and user-space domains is so sharp, there may be several LTTng session and consumer daemons: a session daemon in kernel space and another one in each distinct user-space, as well as one to three consumer daemons in kernel space and another one or two in each distinct user-space (for convenience and efficiency, the consumer daemons exist in 32- and 64-bit versions to accommodate process bitness). See Section 3.4 for a detailed run-down of the possible configurations and their consequences. The simplest, most flexible configuration uses just one session daemon and three consumer daemons (all four of them in kernel space), but this does mean access to the kernel session daemon is a privilege. If there is no need for kernel tracing (e.g. a user wants to debug an application he’s written), an unprivileged user’s needs can be satisfied using just a trio of unprivileged daemons (session and 32- and 64-bit consumers) running in his user-space.

Because accessing the kernel daemon (i.e., sending it commands) normally requires elevation and it is rather inconvenient to prefix all `lttng` commands with `sudo`, LTTng provides the `tracing` group. Under normal circumstances, if the user belongs to the `tracing` group commands sent to the `lttng` program will be passed along to the kernel session daemon (and their answers likewise passed back). In Figure 4, `User1` is unprivileged and uses a local session daemon to trace applications running in his own user-space. He cannot trace anything else. `User2`, by contrast, belongs to the `tracing` group: his local `lttng` commands are redirected to the root session daemon, allowing him to trace any events, be they from the kernel, from his own user-space or from any other user-space.

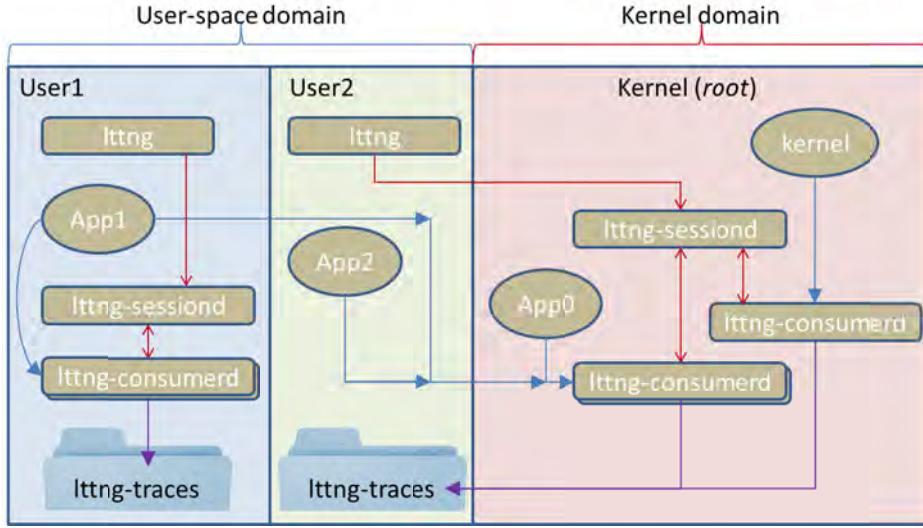


Figure 4: User2 enjoys the benefits of belonging to the tracing group.
User1 can only trace his own user-space. See the Figures of Section 3.4 for the legend.

There is one subtle operational difference between a user belonging to the tracing group issuing “unprivileged” commands (undergoing implicit elevation) and the same user issuing explicitly elevated commands (by prefixing `lttng` with `sudo`). Although in both cases the conversation is held with the same root session daemon and the set of visible events is the same, the session daemon does take note of which user ID is talking to it, *and segregates the sessions accordingly*. Sessions created by a given tracer user (`lttng create whatever`) are visible *only* to that user and root; to see all sessions managed by the root daemon, you must use `sudo` (`sudo lttng list`). Here is an example where two users (of the same system, both belonging to the tracing group and both having `sudo` privileges) are issuing `lttng` commands, each in his own shell (one per column):

```

tracer1$ lttng create ses1u
tracer2$ lttng create ses2u
tracer1$ lttng list
1) ses1u
tracer2$ sudo lttng create ses2r
tracer2$ lttng list
1) ses2u
tracer2$ sudo lttng list
1) ses1u
2) ses2u
3) ses2r

```

This state of affairs means a tracing user will get a ‘session already exists’ error message if he tries to create a session bearing the same name as one created by another tracing user, even though the name does not appear in his `lttng` list.

A Ubuntu installation (see Section 3.1) or DVD-ROM installation (see Section 3.2) creates the tracing group for you. If you build `lttng-tools` instead (see Section 3.3.3.4), you will need to create the tracing group yourself. In all cases you will need to join one or more users to it if you want those users to easily trace the kernel. See Topical Note A.4 for a summary of group interrogation, creation and manipulation commands. Note that uninstalling the DVD-ROM's `lttng-tools` package will delete the tracing group, so if you had joined it and then perform a reinstallation, you will need to join the group again.

From the point of view of the tracepoint providers used to instrument applications, there are two `lttng-sessiond` daemons that need to be registered with: the root daemon and the local daemon. The session daemons themselves aren't aware of each other. This is also true of the consumer daemons, which always report to a single session daemon.

2.3 Flow of control and data

The overall flow of control and data is shown in Figure 5, below. A tracing operation begins with commands issued to the LTTng system in order to define what will be traced, where the resulting trace will be stored, and when to trace (i.e. starting, pausing, stopping). These commands may be sent by the `lttng` command-line client, but can also be issued by any application that implements the LTTng control API (`liblttng-ctl`), such as Eclipse LTTng Tracing and Monitoring Framework (TMF) plug-ins. The latter are beyond the scope of this document; suffice it to say that it should be a relatively simple matter to automate LTTng operations.

Using the `lttng` command-line client, you interrogate the LTTng managers: the `lttng-sessiond` daemons. There may be several of these, one running as root and in charge of kernel tracing, and one in each of the user-spaces in existence on the system (see Section 3.4 for the gory details). The `lttng-sessiond` daemons manage the activation and deactivation of tracing events, and direct the tracers to a dedicated collection of buffers. The `lttng-sessiond` daemons also manage the `lttng-consumerd` daemons during tracing (see below). As explained earlier in Section 2.1.7, LTTng's kernel tracing facilities are implemented by a series of small kernel modules which "service" the numerous tracepoints embedded in the Linux kernel. User-space tracepoints are handled in similar fashion by special-purpose "trace providers".

The forthcoming addition of namespaces to Linux systems may add wrinkles to LTTng tracing; see Topical Note A.5 for details.

Events are committed to the LTTng buffers using a wait-less algorithm that privileges performance over integrity. By design, losing events is acceptable when the alternative would be to cause substantial delays in the instrumented code. LTTng aims to perturb the traced system as little as possible, in order to make tracing of subtle race conditions and rare interrupt cascades possible.

During tracing, another set of daemons, the `lttng-consumerd` instances, manages the extraction of the event records from the buffers and their dispatching to persistent storage. There may be several `lttng-consumerd` daemons: up to three for root (one for the kernel domain and another two for the user-space domains), and up to two for each tracing user-space (for convenience and efficiency, the consumer daemons exist in 32- and 64-bit versions to accommodate process bitness; see Section 2.1.2). Note that the root `lttng-consumerd` daemons will capture user-space events issuing from *any* user-space, whereas a user-space `lttng-consumerd` can only capture that user-space's events (see Figure 27 through Figure 29 in Section 3.4). Note also that the user-space events of root applications (running in root's user-space) can only be captured by the root consumer daemon.

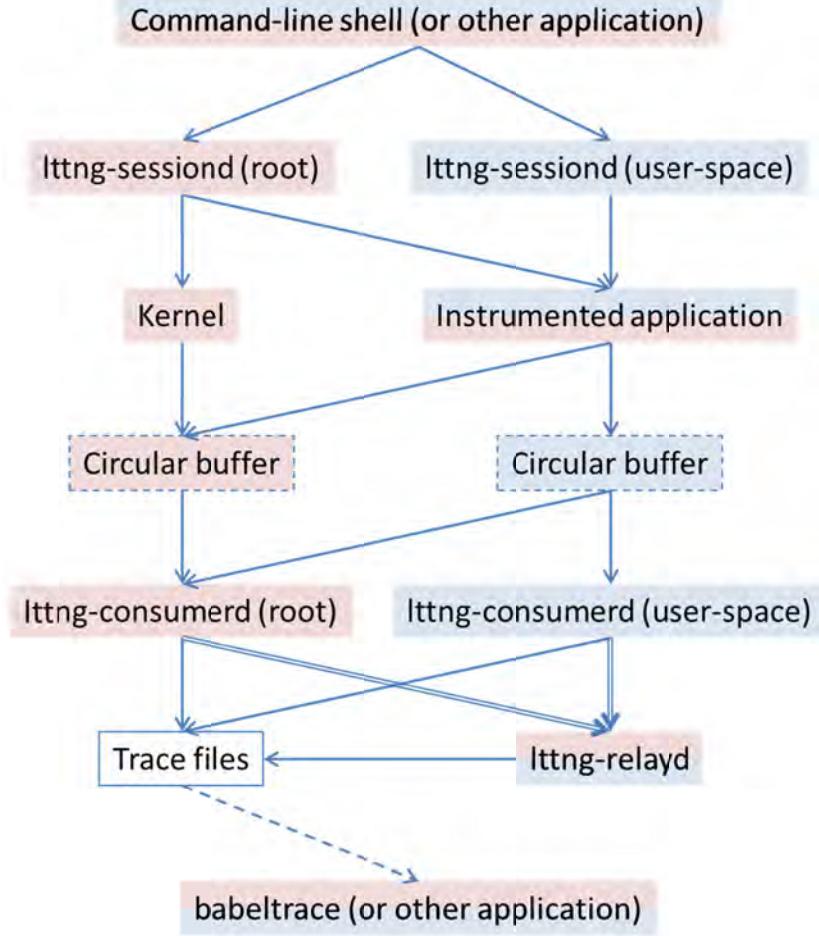


Figure 5: Flow of control and data during an LTTng trace.

Hollow arrows indicate network transport; boxes indicate storage (solid: persistent, dashed: transient). The dashed arrow indicates later processing. Kernel domain processes have a light red background, user-space processes have a light blue background.

Traces may be stored on a local file system, or they may be sent to some remote network destination. In the latter case, another daemon, `lttng-relayd`, receives the event records and deals with them (note that the relay daemon is bitness-independent). A given host's `lttng-relayd` daemon can accommodate event streams coming from several different traced machines. The default `lttng-relayd` daemon stores the event records on its host's file system, but a special-purpose `lttng-relayd` daemon could deal with them in arbitrary ways (e.g. keeping a scrolling buffer of events which is processed into a stream of metrics, etc.).

At some point in the future, `babeltrace` (or any other trace viewer) reads the trace files for visualisation, analysis or any other purpose. Live traces will be made processable in the near future, as has been said earlier.

2.3.1 Discard and overwrite modes

Each buffer in Figure 5 is potentially being accessed simultaneously by a multitude of writers and a single reader (a.k.a. consumer). The potential for conflict is great, but LTTng's architecture minimises it. First, the writers as a group interfere very little with each other. In order to deposit an event record into the buffer, each writer must first reserve a slot of the appropriate size. When two writers attempt this reservation at once, one succeeds while the other fails. The unsuccessful writer simply repeats its attempt to reserve a slot until it succeeds. Once a slot has been reserved, the writer can "at leisure" write its event record into it.

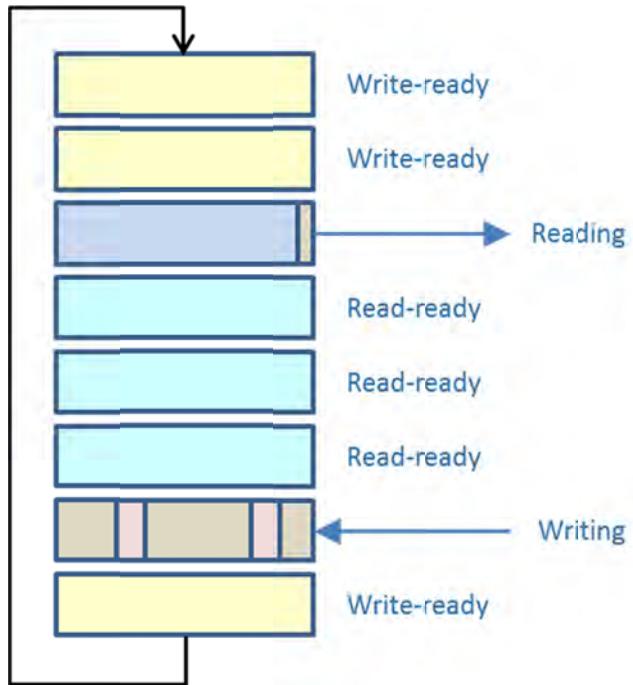


Figure 6: A circular buffer and its individual sub-buffers' states.
The four states (write-ready, writing, read-ready and reading) are mutually exclusive. The consumer reads a whole sub-buffer at once, skipping only the padding bytes, if any.

This leaves the matter of conflicts between the writers and the consumer. To manage this, the buffer is broken down into a number of discontiguous sub-buffers (see Figure 6). The consumer reserves (obtains a lock on) a whole sub-buffer at once, because it intends to copy all of the event records in the sub-buffer to their final destination (a trace file). Once the consumer is finished with the sub-buffer, it becomes available for re-use by the writers. If the consumer outperforms the writers, it will eventually come up behind them and stop just before the last sub-buffer being written into. At this point, the consumer will simply wait until the writers are finished with the sub-buffer. It is acceptable for the consumer to block in this way, as this does not compromise the trace's contents nor the system's performance.

When the writers outperform the consumer, the situation is different. By design, the writers cannot block, because this could make the entire system freeze. One need only think of a trace capturing an oft-used kernel tracepoint, such as memory allocation. If the kernel blocked within such a tracepoint, every application running on the system would very quickly block in turn, waiting for the kernel to resume. In order for the writers to remain wait-less, event loss becomes unavoidable. This can happen in two ways.

When the trace operates in *discard mode*, a writer that cannot obtain access to the current sub-buffer (because the buffer is full, whether or not a consumer is busy processing the current (oldest) sub-buffer) simply discards its event record and continues. The trace tracks how many events are lost in this way. This mode may also be called *upstream* or *early loss*, because the event record is lost very soon after its occurrence. Discard mode is the default operating mode (see the `discard` channel option of Section B.4.8).

The alternative is *overwrite mode* (see the `overwrite` channel option of Section B.4.8). In this mode the writers never pause: they freely grab sub-buffer after sub-buffer and write into them. Meanwhile, the consumer uses a separate, spare sub-buffer which is swapped in and out of the ring of sub-buffers (see Figure 7). This way the writers never have access to the consumer's sub-buffer and thus cannot overwrite its contents while the consumer is busy reading them. When the consumer is done with a sub-buffer, it simply requests the next sub-buffer from the ring (the buffer management data structures include a sub-buffer index number which can be understood as a sort of timestamp).

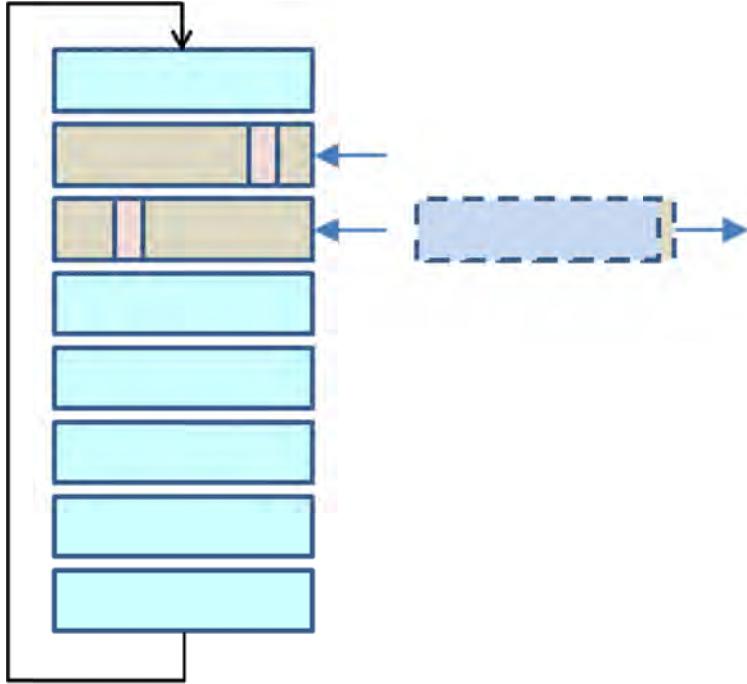


Figure 7: A circular buffer in overwrite mode, showing the extra sub-buffer.

The consumer swaps its sub-buffer out of the ring (dashed box) so it can process it without interference. The write-ready and read-ready states are now indistinguishable: it's just a matter of which process, between the reader and the writers, gets to those sub-buffers first.

If the sub-buffer is available (i.e. has not been overwritten yet), the spare sub-buffer is atomically swapped with it, and the consumer can then proceed with its task. If the sub-buffer is available but busy (writers are still writing into it), the consumer simply blocks until the sub-buffer becomes available, just like in discard mode. Finally, if the sub-buffer is wholly unavailable (i.e. it has been overwritten or is in the process of being overwritten), the consumer skips ahead to the next sub-buffer, which will eventually resolve to either of the two preceding cases. The trace merely tracks the time-slices that are lost in this way. This mode is also called *flight recorder mode*, and may also be called *downstream* or *late loss*, because the event record is lost after having been kept in the buffer for a while.

Each writer is “in” the buffer only sporadically, from the time it reserves a slot in a sub-buffer until it has completed writing its event record and releases the sub-buffer slot. While a writer is “in” a sub-buffer, the latter is *dirty* (tawny colour in Figure 6 and Figure 7). Should a writer die or freeze during that time, the sub-buffer becomes *corrupt*. In either case it is unreadable. In discard mode, the consumer is always ahead of the pack of writers (since the writers cannot pass it), so it will eventually get around the ring of sub-buffers and come back to the unreadable sub-buffer. In overwrite mode, it can be either the consumer or some other writer that comes around to the unreadable sub-buffer first. In both modes, when this happens the sub-buffer is dismissed and marked as ready for re-use. The trace will include a time gap, just like in the usual overwrite event loss case. If the writer that was “in” the dismissed sub-buffer later resumes execution, its write operation will simply fail. The count of lost events is not increased in this case, because it is stored in the now-dismissed sub-buffer.

There is a third, intentional event loss mode: setting both the channel options `tracefile-size` and `tracefile-count` to positive values transforms the trace folder into an on-disk flight recorder (see Section B.4.8).

2.4 Types of probes that can be used with LTTng

LTTng is able to capture several types of trace events: tracepoints (the main kernel and LTTng tracing facility), system calls (`syscalls`), kprobe and kretprobe events (see also Topical Note A.6). LTTng also has access to CPU performance monitoring unit (PMU) counters (`perf`). Some infrastructure is in place for ftrace and uprobe events, but they are not ready for exploitation yet. The ultimate objective is to allow any tracing solution to write its events into an LTTng trace. For instance, the Linux kernel's to-do list still includes user-space probes and watchpoint probes (probes that can be attached to data symbols) [10]; LTTng will naturally incorporate these into its probe set as soon as they become available.

Not all probes may be available, depending on the kernel's configuration. There are currently four (ultimately three) kernel configuration options which are absolutely required for LTTng to be functional [4] [11], one of which is `CONFIG_TRACEPOINTS` (see Section 3.3.3.1 to see how to check your kernel's actual configuration). Tracepoints are thus always available as long as LTTng is installable. The remaining probe types described here each depend on a specific kernel option being turned on. If turned off, you will get an appropriate error message when you try to enable the unavailable event type.

- ◆ `CONFIG_HAVE_SYSCALL_TRACEPOINTS` is required in order to trace system calls (the `syscall` option of `enable-event`, Section B.4.10).
- ◆ `CONFIG_KPROBES` is required in order to use kprobes (the `probe` option of `enable-event`).
- ◆ `CONFIG_KRETPROBES` is required in order to use kretprobes (the `function` option of `enable-event`).
- ◆ `CONFIG_EVENT_TRACING` converts the `blktrace` (block input-output) kernel tracer events into LTTng tracepoints events.

A fifth kernel configuration option of interest is `CONFIG_PERF_EVENTS`, which is required in order to use `perf` in a channel context (the `add-context` command, Section B.4.1).

2.4.1 Tracepoints

Tracepoints are strategically located throughout the kernel; they are static strictly-typed functions that can be activated and deactivated dynamically. They have a very low performance cost when inert, and are designed to be interruptible and re-entrant. They also have nearly universal portability compared to, for instance, `syscalls` (which require some architecture-specific code in the kernel and are thus not available on all platforms).

They take the form of jumps to function calls to tracepoint providers, guarded by immediate value test instructions. This allows the tracepoint to be enabled and disabled atomically, by changing the immediate value byte in the code.

2.4.2 Perf

Perf is a hardware and software per-process counter sampling facility. It is mostly geared to the kernel, although user-space performance monitoring is possible albeit at a significant performance cost (because of context-switching) and with limited unprivileged user control. Tracing was added using the infrastructure developed for sampling. Its event header is unique and has a fixed format, although it allows payload definition extension. In development environments, `perf` can give insight into the major bottlenecks slowing down process execution, when the cause of the slowdown can be pinpointed to a particular set of processes [4].

LTTng uses only the sampling capabilities of `perf`, by allowing `perf` counters to be added to channel contexts (see the `add-context` command, Section B.4.1).

2.4.3 Kprobes

KProbes are transient dynamic probes that instrument specific kernel addresses by replacing them with an interrupt that leads to the KProbe manager [12]. KProbes, like ftrace, can be controlled using the debugfs-supplied `/sys/kernel/debug/tracing` folder [13]. UProbes use similar mechanisms (including control from the `/sys/kernel/debug/tracing` folder) to instrument user-space applications. Unlike LTTng's user-space tracepoint instrumentation, uprobes run in the kernel space like kprobes [14].

LTTng's implementation of kprobes captures a single payload field: `ip`, the value of the processor's instruction pointer (IP) —a 32- or 64-bit hexadecimal integer, depending on your processor's bitness. This will always be the same value, since it marks the address where the kprobe was inserted. It is possible to repeatedly kprobe the same address: the kprobes will stack, most recent first.

LTTng's kprobe events are tracepoints in the sense that LTTng supplies a thin wrapper that acts as a tracepoint provider. The kprobe events are controlled (enabled, disabled, assigned to a channel) like other kernel-domain tracepoints. At the same time, they are *not* tracepoints, in the sense that they use the kprobe trampoline insertion mechanism instead of the tracepoints' lighter guarded function jump mechanism. This dichotomy is apparent when you consider that one could, in theory, install kprobes outside of LTTng's scope, and LTTng wouldn't be aware of those events.

2.4.4 Kretprobes

KRetProbes are an extension of KProbes; they serve to instrument the return from a target function [15]. LTTng's `enable-event --function` command (see B.4.10) handles kretprobes by setting up *two* events at the indicated address or symbol: an `*_entry` event and a `*_return` event [4]. For instance, setting up a kretprobe like this:

```
$ lttng enable-event sysopen -k --function sys_open+0x00
```

Will generate the events `sysopen_entry` and `sysopen_return`. To the `ip` payload field is added a second field, `parent_ip`, which is unfortunately completely undocumented. If kretprobes are stacked at the same address, the entry events will occur most recent first, but the return events will be least recent first: the kretprobes are nested.

2.4.5 Syscalls

System call tracing is complementary to kernel tracepoint tracing.

System calls, in Linux, are the gateways into the kernel. They are not exposed directly to programs; instead, the system API includes a number of *system call wrappers* which do a little bit of parameter preparation before invoking the actual system calls. Some wrappers end up invoking the same underlying system call (for example, `malloc()`, `calloc()` and `free()` all call `brk()`). The system calls are compiled as software interrupts (interrupt 128 (0x80)) or the interrupt-like `sysenter` instruction (introduced with the Pentium II) [16]. A system call's arguments are stored in the processor's registers in order to survive the context switch, and the interrupt (or `sysenter`) handler, the *system call handler*, takes care of redeploying the register contents to the kernel's stack. The *system call's number* (passed in the EAX register) is then used as an index to look up the address of the *system call's service routine* in the *system call dispatch table* (`sys_call_table`). The system call service routine is then resolved as a normal C function. Once it concludes, the system call handler reverses the process, storing the stack in the registers and switching the context back to user-space.

The important point to retain is that system calls are all handled by the same kernel routine, which merely dispatches the call according to its number. The actual service routine any given system call ends up in can vary dynamically, in particular when loading a kernel module changes an entry in the system call dispatch table.

LTTng's system call tracing facility (the `lttng-tracer.ko` LTTng kernel module) instruments the kernel's system call handler and converts the system call entry events into specific tracepoint events (instead of a single generic `sys_entry` event). Because system calls are never nested, the system call exit event remains common to all system calls and is merely renamed from `sys_exit` into `exit_syscall` to avoid the confusion with the homonymous `sys_exit` system call (by convention, system call events are named by prefixing the system call with `sys_`; thus `brk()` will show up as a `sys_brk` event, and `exit()` will show up as a `sys_exit` event).

Not every system call is defined in the kernel (or its modules) with a `SYSCALL_DEFINE` macro, so some system calls are missing in `kallsyms` (during its build, the kernel creates a `__kallsyms` data blob section that lists all of its non-stack symbols, complementing the list of symbols exported to kernel modules; this allows debuggers and the like to look up any kernel address and find out the owning kernel or module, the containing section within the owning code, or the nearest symbol). These “missing” system calls are shown as `sys_unknown` by LTTng [17]. Examples for ARM include `sys_recv`, `sys_sigreturn`, and several others. This situation tends to resolve itself as kernel description files are fixed, and LTTng already accounts for most such “missing” syscalls. Thus, `sys_unknown` occurrences in a trace are likely only when using a kernel that is newer than the LTTng installation.

There are about as many system calls as there are kernel tracepoints: nearly 300, although this number varies somewhat with the underlying architecture and the kernel version. Depending on the compilation options, quite a few of the system calls may end up serviced by the `sys_ni_syscall()` routine, which merely reports the system call as “not implemented”.

On 64-bit systems, there is a second set of 32-bit retro-compatible system calls, accessed through a separate dispatch table, that are used by 32-bit applications. The corresponding system call event names are further prefixed by `compat_` (e.g. `compat_sys_brk`, `compat_sys_exit`, `compat_sys_ni_syscall`, `compat_sys_unknown`). The `exit_syscall` event keeps the same name [18].

A comparable user-space function tracing facility is supplied by the `liblttng-ust-cyg-profile` libraries. See Annex J for the details.

See Topical Note A.7 to learn how to manually update LTTng’s set of system call names (after updating or modifying the kernel, for example).

2.5 Trace events

Event names are arbitrary (user-defined) for certain types (`probe`, `function`) but not for others (`syscall` and kernel tracepoint names are system-defined, application tracepoint names are defined by their source code). The span of user-defined event names is not a settled issue at this point; it may be that eventually the same name could be used for different events in different channels, domains or sessions, but some combinations cause problems in the current LTTng implementation and such a practice should be avoided. It muddles what is meant by any given event name anyway, making comparisons between traces difficult.

In the kernel domain, you should avoid using a pre-defined event name (kernel tracepoints or system calls (e.g. `sys_open`, `sys_sched_yield`, etc.), including the LTTng-reserved names `exit_syscall` and `sys_unknown`) for the arbitrary-named event types. Not doing so can lead to shadowing (see Section B.4.10) or worse.

With user-space tracepoint providers, care should be taken to avoid provider name collisions and when managing successive revisions (you should make sure to delete, archive or rename older tracepoint provider dynamic libraries so the correct version is loaded by the instrumented clients). Even the tracepoints themselves are known [19] to occasionally give rise to naming conflicts. This is a problem for the generated trace which cannot tell the two tracepoints apart if they originate from the same domain; since they may have completely different payloads this may be a show-stopper for some analysis applications.

Each event has a well-defined set of data that it will capture: the trace records, for each event, its timestamp, origin (which processor, user and process ID), identification (event type), and payload. LTTng can freely enrich all events of a given channel with context information (additional data for the record), including `perf` counter data (see Section B.4.1). The CTF specification actually allows event traces to lack timestamps (or even event IDs): the events are nevertheless ordered by their apparition sequence within each trace file (and would all be of the same unspecified event type). This is an unusual configuration which LTTng does not produce.

A word on timestamps: The notion of *when* an event occurs is not as simple as it may initially appear. The instruction pointer of a given processor reaches a given address during a specific clock tick, but generating the trace event itself is not instantaneous: it involves a number of instructions. The earliest a timestamp can be committed would be when the trace provider is first invoked and starts to resolve the event. The trace provider gathers contextual data (the event’s payload), including some additional context the user may have tacked on to the event (see the `add-context` command, Section B.4.1; context is a configuration option similar to event activation —the session daemon simply turns it on within the provider [20]). Once this is complete, a slot is reserved in the tracing buffer, and the event record is copied to it. In order to preserve the assertion that successive events in a buffer are in strictly increasing timestamp order, it turns out the timestamp is assigned by the tracer to the event at a “late” stage, specifically *when the buffer slot reservation is done*. In Figure 5, this corresponds to the arrowhead entering either dashed box. This has the counter-intuitive effect that a single event to which several traces subscribe will bear a different timestamp in each trace, because each event record instance may reserve space in each buffer at a different moment, even though it is written by the same tracer. Keep this in mind when trying to reconcile different traces of the same system time slice.

Tracing sessions do not impinge on each other even when managed by the same session daemon. There is one exception: because of the underlying mechanism, extrinsic probes such as the kprobe event type are bound to interfere somewhat with each other, even between different session daemons. Also, multiple sessions tracing the same events will affect each other’s performance slightly.

2.6 LTTng output files

2.6.1 The concept of “channel”

If tracing were controlled strictly at the event type level, it would be somewhat like having a filing system that consisted of a single list of files. A filing system that adds the concept of folders is much easier to manage and to use. In similar fashion, LTTng uses *channels* to organise event streams both in logical terms within the resulting traces and in operational terms during tracing.

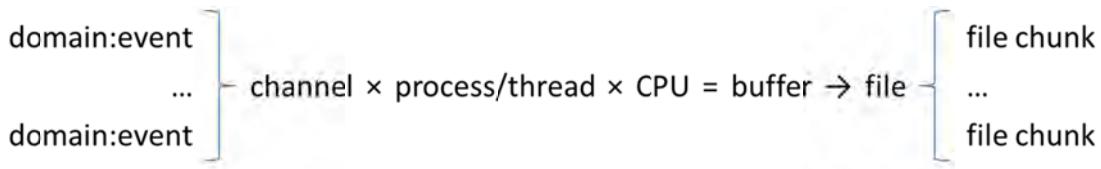


Figure 8: Channels and buffer multiplicity for the per-process-ID buffering scheme.
The per-user-ID buffering scheme substitutes “user” for “process/thread”.

Figure 8 illustrates the role of channels within traces. Event types are defined within a given domain, and channels are arbitrary sets of event types (all taken from the same domain). The events of each channel can occur in a variety of processes or threads, and —because of task scheduling and task migration— each occurrence can happen on any one of the CPU cores. Each channel–process–CPU combination has a dedicated memory buffer (in the per-user-ID buffering scheme, the owning user ID substitutes for the individual process IDs, reducing the number of buffers required while conversely increasing their size requirements). The contents of each buffer are committed to a separate trace file. Optionally, each trace file may be chunked over time. Each buffer is in turn realised as a (circular) set of sub-buffers, and the consumer daemons transfer each sub-buffer as a single *event packet* consisting of the event records held in the sub-buffer.

This multiplicity is the “worst case” and occurs only with per-process-ID user-space traces (see the example in Section 2.6.2.2 and the channel options in Section B.4.8). Per-user ID user-space traces merge the process event streams based on the user ID they belong to, and thus have lesser multiplicity. Kernel traces, finally, are essentially single-user (user ID zero, a.k.a. root) traces — the channels are only multiplied by the number of CPU cores.

Event type instances (event occurrences) issuing from a given source (application, kernel module, etc.) form *event streams*, one per event type. On multiple-CPU platforms, each event stream subdivides into as many streams as there are CPUs; in this document these are called *event stream bundles* (shown by the magnifying glass in Figure 9 below). Recall that, in most cases, any process or thread may migrate between CPUs at any time. The actual event record preparation (payload collection) and send-off is handled by a small code module, the *tracepoint provider* (typically a dynamic library for applications, a kernel module for kernel event types; see Section 2.1.7). Each provider may handle one or more event types, generating one or more event streams, and each event source (instrumented module) may call upon one or more providers. In the example of Figure 9 below, applications App1 and App3 both use the same provider to generate event type ev1.

A single provider could be linked differently into two different applications (see Section 4.1.3). For example, App1 could use a statically-aware dynamic link while App3 could be statically linked or, conversely, use a fully dynamic link (an explicit `dlopen` instruction). The important thing is that, should the provider change (adding, removing or renaming event types, changing the event type arguments, or modifying the event type fields; see Section 4.1.2.2), these changes be properly propagated. Statically linked instrumented clients will certainly need a recompilation, while dynamically linked ones will need it only for certain types of changes. One should also be careful that tracepoint providers realised by multiple instances be functionally identical (have the same signatures, payloads and internal processing).

Figure 9 below shows each kernel event type issuing from a different kernel functional area (a “module” in the loose sense of the term). Although it is usual for kernel functionalities to be regrouped in modules (all functions dealing with task scheduling, for example) and for event types relating to these functionalities to bear same-prefix names (e.g. the `sched_*` kernel event types), LTTng is quite agnostic on this point: kernel event instances issuing from any point in the kernel are dealt with according to their type, period.

Tracing sessions are completely independent of each other and may run concurrently. The only requirement is that their names be globally unique. Within each session's domains, *channels* are created which regroup event stream bundles. Channel names are arbitrary (user-defined) just like session names. Each event type can be assigned to one or more channels (kernel event types are currently restricted to one channel at a time, however). Channels can be enabled or disabled dynamically, effectively turning on and off the event streams. Each channel's event subscription may be subjected to a different filter (see the `enable-event filter` and `loglevel` options, Section B.4.10), so you could for instance exclude (filter out) event occurrences that have a certain span of user IDs from a channel.

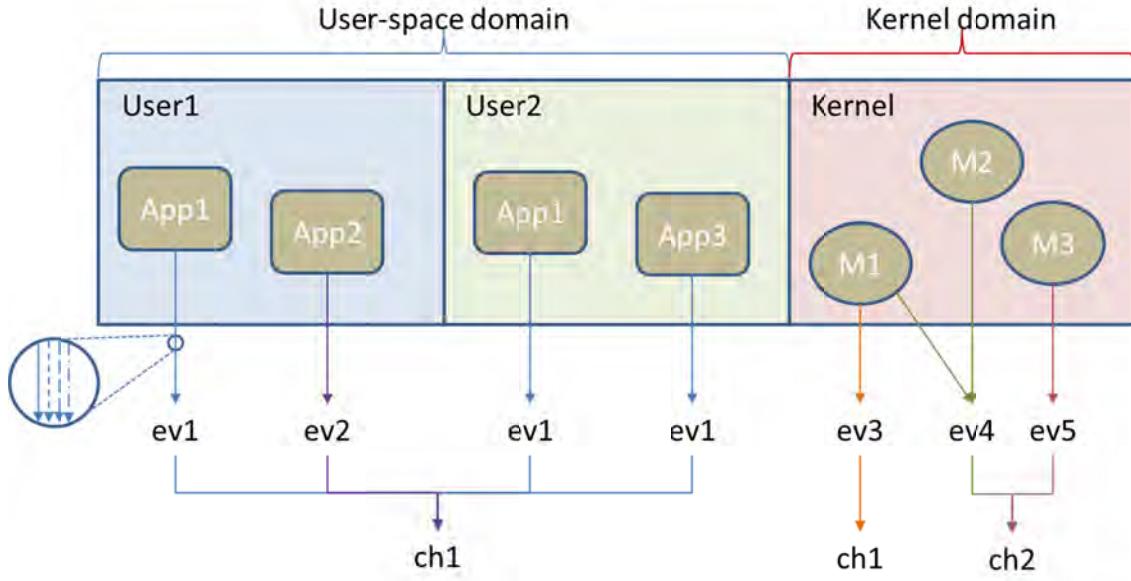


Figure 9: Channel structure of a tracing session.

The magnifying glass in the lower left shows how each event stream is actually an event stream bundle, with one sub-stream for each CPU (here a four-CPU machine is assumed).

In Figure 9 above, there is one user-space event stream bundle of type `ev2`, and three of type `ev1`, both assigned to the same user-space channel, `ch1` (the various `ev1` stream bundles could be separated using filters if one wished to do so). In the kernel domain, the event types `ev4` and `ev5` are assigned to channel `ch2`, while event type `ev3` is assigned to channel `ch1`. The same channel name was used to stress the fact that channels in one domain are completely independent of those in another domain. The kernel events can issue from the kernel itself or from kernel modules (as shown here). Root applications (typically daemons) can also generate events but they are considered user-space events, albeit belonging to root's user-space and thus requiring kernel tracing privileges (see Section 3.4.1 for examples).

While a session is running and a trace is being captured, channels are the main dynamic configuration mechanism, as they can be disabled and enabled quickly and efficiently. It is also possible to reconfigure events (enabling, disabling, filtering) while the trace is running. As of LTTng 2.3.0, channels cannot be added while a trace is running (be they user-space or kernel). In both domains each channel is immutable once created as far as its configuration goes, but events can be added and removed (disabled) and the channels themselves enabled and disabled on the fly. Some of these restrictions may be lifted in future versions of LTTng.

The internal configuration of the tracer is also specified at the channel level. As the various options of the `enable-channel` command show (see Section B.4.8), this is where you choose the size, partitioning and multiplicity of the buffer (i.e. whether to organise trace files per process or per user), whether to chunk the resulting trace files, how to handle event overflow (see Section 2.3.1), etc. For instance, in the Figure 9 example, the choice is between per-process-ID or per-user-ID (the default) file partitioning. Assuming a single-processor system, in the former (per-process-ID) case, the event streams coming from `App1` (both `User1`'s instance and `User2`'s), `App2` and `App3` would each go through a separate circular buffer and ultimately be stored in four separate files. In the latter (per-user-ID) case, there would be only two buffers and as many files for that channel. If the system were to have a quadruple core (for instance), each event stream would become a four-stranded event stream bundle and the buffer and file counts would be multiplied by four accordingly.

2.6.2 Output files and folders

2.6.2.1 Files

The trace records end up stored in a hierarchy of folders and files, summarised in Figure 10. This hierarchy is rooted in the *trace root folder*, `$HOME/lttng-traces` (unless specified otherwise). Local traces are created there, while remote traces (traces received over the network by the `lttng-relayd` daemon) are created in subfolders named for the hosts. For example, suppose there is a local trace named `can-20130611-112233` and a remote trace named `usa-20130611-122334` coming from a host named `america`. Then the trace paths would be respectively:

```
$HOME/lttng-traces/can-20130611-112233  
$HOME/lttng-traces/america/usa-20130611-122334
```

The trace names, as explained in Section B.4.3 (the `create` command), consist of the session names suffixed with a timestamp. This allows users to re-use tracing session names over time without fear of overwriting previously saved trace data. Using the `output` option of the `create` command, the user is free to set the *session folder* path to an arbitrary destination. For example, instead of `$HOME/lttng-traces/can-20130611-112233`, one could set the session folder path to `/usr/share/nafta`.

It is possible to introduce spurious intermediate folders in the trace path by using the folder separator in the session name. For example, one could create a session named `can/usa`, which would obligingly be written to `$HOME/lttng-traces/can/usa-20130611-112233`.

Because snapshots (see Section B.4.13) are whole traces by themselves and may occur multiple times with a session (by contrast, non-snapshot mode tracing sessions each generate a single trace, albeit broken up into multiple trace folders), they insert a further subfolder layer. If the `can` and `usa` example traces were to consist of snapshots instead of being single traces, the paths would be something like:

```
$HOME/lttng-traces/can-20130611-112233/snapshot-1-20130611-113300-0  
$HOME/lttng-traces/can-20130611-112233/snappy-20130611-114400-1  
$HOME/lttng-traces/america/usa-20130611-122334/snapper-20130611-122400-0
```

Example

Suppose `userAlex` runs LTTng to trace his machine (storing the trace locally):

```
$ sudo -H lttng create lsession  
$ sudo -H lttng enable-event sched_switch -k  
$ sudo -H lttng start  
$ sudo -H lttng destroy
```

The session folder path will be `/root/lttng-traces/lsession-20130401-120000`. Note that this folder is owned by root because it was run as root using `sudo`, and that it is stored in root's home folder (`/root`) because `sudo -H` was used.

The next trace run by `userAlex` is for `userBeatrice`, whose IP address is 131.132.32.66. All `userBeatrice` has to do beforehand is to run `lttng-relayd -d` on her machine:

```
$ sudo -H lttng create rsession -U net://131.132.32.66  
$ sudo -H lttng enable-event sched_switch -k  
$ sudo -H lttng start  
$ sudo -H lttng destroy
```

That session folder path will be `/home/userBeatrice/lttng-traces/Alex/rsession-20130401-130000` on `userBeatrice`'s machine. The network name of `userAlex`'s machine is `//Alex`, hence the `Alex` folder in `userBeatrice`'s `lttng-traces` folder. Note that because `lttng-relayd` was run as an unprivileged daemon, it stored the trace in its user's `lttng-traces` folder. Note that the session folder is chosen by `lttng-relayd`: there is no way for `lttng-sessiond` to request that `lttng-relayd` use another path.

Whereas the `lsession-20130401-120000` session folder is created as soon as the `enable-event` command is issued by `userAlex`, the `rsession-20130401-130000` folder is created only once the `start` command is issued by `userAlex`.

2.6.2.2 Folders

Within each session folder, there may be one or two folders bearing the domain names. For now, these can be `kernel` or `ust`. As mentioned elsewhere, other domains could eventually be added to LTTng. Obviously, the `kernel` folder will only exist when tracing the kernel (i.e. it can only be created by the kernel consumer daemon).

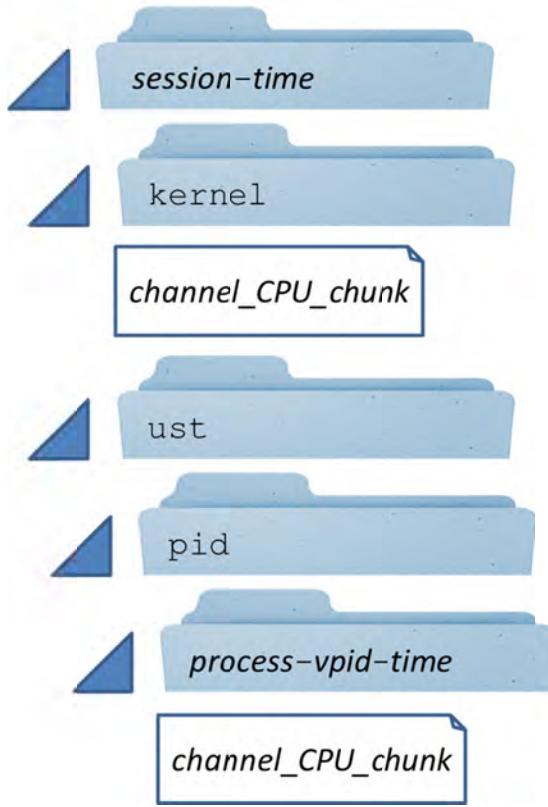


Figure 10: Folder and file structure of a typical stored trace.

One or both of the domain (kernel and ust) subfolders may be present. This is a local trace using per-process-ID user-space buffering. Strings in italics (session, channel, CPU, chunk, process, vpid, time) are place-holders.

```
$HOME/lttng-traces/can-20130611-112233/
  kernel/
$HOME/lttng-traces/can-20130611-112233/
  ust/pid/synaptic-2486-20130611-122334/
```

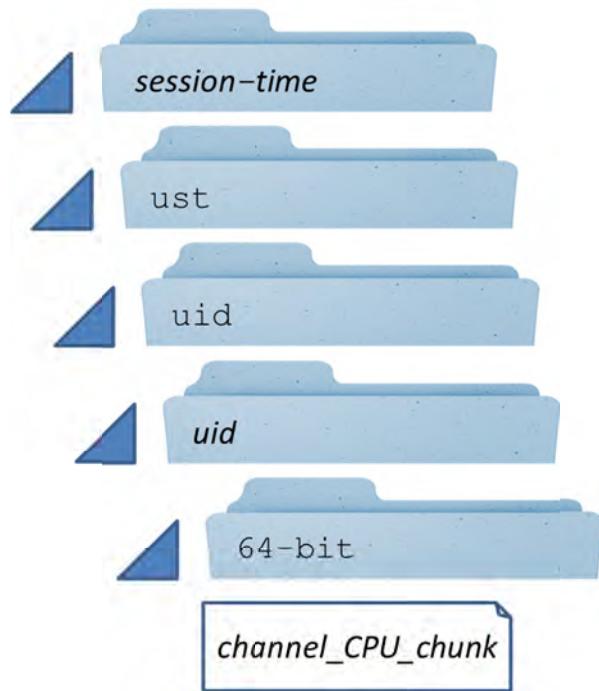


Figure 11: Folder and file structure of a trace using per-user-ID buffering.
This trace has no kernel domain for clarity.

```
$HOME/lttng-traces/usa-20130612-102030/  
ust/uid/1000/64-bit/
```

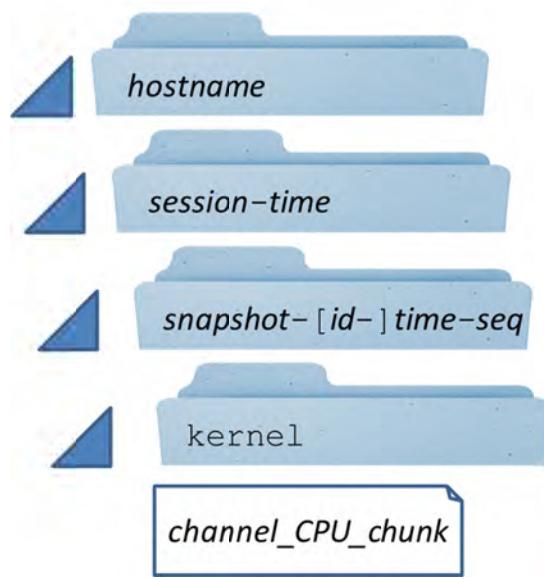


Figure 12: Folder and file structure of a remotely stored trace (coming from hostname). This trace has just the kernel domain and runs in snapshot mode (multiple snapshot subfolders may populate the session folder).

```
$HOME/lttng-traces/namerica/mex-20130613-142536/
    snapshot-0-20130611-122334-1/kernel/
```

The `kernel` folder has no subfolders: kernel tracing creates a single *trace folder*. The `ust` subfolder, on the other hand, will contain either a `pid` subfolder or a `uid` subfolder³. The first of these will contain subfolders named according to the following scheme: “`<name>-<vpid>-<timestramp>`”, where `<name>` is a process name (truncated to 15 characters by the system), `<vpid>` is a virtual process ID (process ID within a PID namespace, see Topical Note A.5), and `<timestramp>` is constructed just like the session’s, except that it matches the time when the first events are recorded for that particular process. As tracing may be suspended and resumed, a given `<name>-<vpid>` combination may recur (this will happen for example if a process manages its own traceability by dynamically loading and unloading its trace provider).

In the `uid` case, the subfolders are simply named `<uid>` (that is to say, they are labeled with the numeric user ID). They contain a 32-bit subfolder and/or a 64-bit subfolder, depending on the bitness of the captured events. Configuring LTTng so that both 32- and 64-bit user events can be accommodated (on a 64-bit system, presumably) is a little bit complicated; see Section 4.5 for the details.

³ Channel buffering schemes may not be mixed within a session. It is however possible to create two different sessions sharing a single session folder: as long as at most one of the two traces the kernel and the two sessions use different user-space channel buffering schemes, no collisions will occur.

Examples of fully developed *trace folder* paths (leaf folders) are:

```
$HOME/lttng-traces/can-20130611-112233/
    ust/pid/sample-16741-20130611-112244
$HOME/lttng-traces/america/usa-20130611-1223344/
    kernel
$HOME/lttng-traces/america/usa-20130611-1223344/
    ust/uid/1000/64-bit
```

Finally, in the leaf folders are found the trace files. There will be a `metadata` file, which describes the folder contents (event types, host name, domain, tracer name and version, type of clock, etc.), and a number of files named according to the following scheme: `<channel>_<cpu>_<chunk>`. The `<channel>` is the channel name while `<cpu>` is the event stream's CPU identifier; the `<chunk>` is optional and serves to number file chunks when necessary (see the `tracefile-size` and `tracefile-count` options of the `enable-channel` command, Section B.4.8). For example, a typical unchunked trace on a quadruple-core system could have the files `channel0_0`, `channel0_1`, `channel0_2` and `channel0_3`.

The user-space output folder structure was different before the 21 March 2013 7972aab commit because all tracing was implicitly per-process-ID before then. The `pid` subfolder was omitted, the process subfolders appearing directly below the `ust` subfolder.

2.7 Transient LTTng working files

LTTng creates a few transient, hidden files. The already-mentioned `.lttngrc` file is the only one created by the `lttng` client. It is found in the client's `HOME` and holds the client's “current session”. The session daemons maintain a temporary folder as long as they are running. It holds `lttng-sessiond.pid`, which the `lttng-consumerd` daemons use to fetch the process ID of their session daemon, and the sockets `client-lttng-sessiond`, `health.sock`, `lttng-ust-sock-5`, `kconsumerd/error` (root daemon only), `ustconsumerd32/error` and `ustconsumerd64/error`. The root session daemon creates this folder at `/run/lttng`, whereas the user session daemons create it at `$HOME/.lttng`.

Table 1: LTTng transient files.

Name	Location	Significance
.lttngrc	\$HOME of each session daemon	Current session name
lttng-sessiond.pid	/run/lttng (root session daemon) \$HOME/.lttng (user-space session daemon)	Process ID of the session daemon, read by consumer daemon
client-lttng-sessiond	/run/lttng (root) \$HOME/.lttng (user-space)	Client-session daemon communication socket
health.sock	/run/lttng (root) \$HOME/.lttng (user-space)	lttng-health-check socket
lttng-ust-sock-5 [†]	/run/lttng (root) \$HOME/.lttng (user-space)	Application registration socket
command error	/run/lttng/kconsumerd (root)	Kernel consumer daemon command and error sockets
command error	/run/lttng/ustconsumerd32 (root) \$HOME/.lttng/ustconsumerd32 (user-space)	32-bit user-space consumer daemon command and error sockets
command error	/run/lttng/ustconsumerd64 (root) \$HOME/.lttng/ustconsumerd64 (user-space)	64-bit user-space consumer daemon command and error sockets
lttng-ust-wait-5 [†]	/dev/shm/	Kernel shared memory
lttng-ust-wait-5-<uid> ^{†‡}	/dev/shm/	User-space shared memory

[†] The trailing “5” matches the major version number of LTTng’s binary interface.

[‡] The trailing <uid> is the user ID of interest, including “0” for root.

This page intentionally left blank.

3 Installation, testing and controls of LTTng

The first step is, of course, installing LTTng. Linux kernels are already instrumented for LTTng tracepoints as well as for several other tracing facilities (more recent kernels may have more tracepoints than older ones), but this instrumentation is inert, dormant. Installing LTTng adds kernel modules that will service groups of tracepoints, and the LTTng tracing infrastructure itself.

LTTng is a still-evolving software project, so you are faced with a choice:

- ◆ Install the pre-compiled (but obsolete) packages from the Ubuntu repositories (Section 3.1); or
- ◆ Fetch the source code, then compile and install it yourself (Section 3.3); or
- ◆ Install the pre-compiled packages from the LTTng PPA (Personal Package Archive) software repository (Topical Note A.8); or
- ◆ Install the packages from the software repository found on the DVD-ROM which accompanies this document (Section 3.2). (Recommended; this is also the method assumed by the *LTTng Quick Start Guide*)

The four ways differ mostly by date: the Ubuntu packages are the oldest, followed by the LTTng PPA and then the DVD-ROM's software repository. The source code tarballs are always the most recent, of course. As time elapses, expect the DVD-ROM's software repository to eventually slide to the “oldest” position, although this process may literally take months. Sporadic updates of the DVD-ROM's software repository will most likely be released by the author for the near future (by e-mail, considering the relative compactness of the software repository). The work flow choices are illustrated by Figure 13, below.

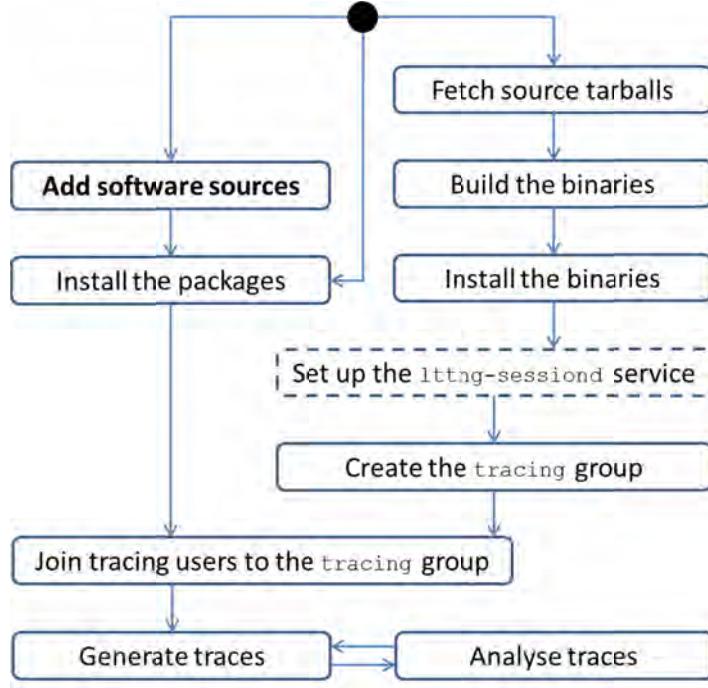


Figure 13: Overall LTTng work flow.

The central branch installs the Ubuntu packages (Section 3.1); the right-hand branch fetches the source tarballs, builds and installs the binaries (Section 3.3) and completes the system configuration; the (recommended) left-hand branch installs from the LTTng PPA (Topical Note A.8) or the DRDC repository (Section 3.2). The dashed step is optional.

Installing from source code (Section 3.3.3) is a little more work and takes a little longer, but does have the advantage of yielding the most recent version with more features, improved performance, and less bugs. Occasionally, the “less bugs” bit will be false, a development release introducing some unforeseen problem. When this happens, report the bug (on the <http://bugs.lttng.org> Web site, or through the mailing list, lttng-dev@lists.lttng.org; see <http://lists.lttng.org/cgi-bin/mailman/listinfo/lttng-dev> to browse its archives or subscribe) and wait. A later release will usually fix the problem in a matter of hours to days.

It is strongly recommended to install from the DVD-ROM’s software repository, if only because most of this document is geared to its 2.3.0 (2013-Sep-06) release as opposed to the current Ubuntu 12.04 LTS (Precise Pangolin) repository release, LTTng 2.0.1 (2012-April-20) (`lttng-tools` version number). The Ubuntu releases and the `lttng-tools` version in their respective repositories are listed in Table 2, below. Earlier LTTng releases had a command-line interface which is by now considerably different from the 2.3.0 interface documented here. The LTTng PPA focuses on stable releases and is currently offering LTTng 2.1.x releases. (<http://archive.ubuntu.com/ubuntu/pool/universe/l/lttng-tools/>)

Table 2: LTTng versions held in the Ubuntu repositories.

Ubuntu version	Release date	lttng-tools version	Date
12.04 LTS (Precise Pangolin)	April 2012	2.0.1-0u1	2012-April-20
12.10 (Quantal Quetzal)	October 2012	2.0.3-0u1	2012-July-16
13.04 (Raring Ringtail)	April 2013	2.1.1-2	2013-April-11
13.10 (Saucy Salamander)	October 2013	2.1.1-2u1	2013-May-07
14.04 LTS (Trusty Tahr)	(April 2014)	2.3.0-2	2013-Dec-03 [†]

[†] As of January 22, 2014.

This document occasionally mentions the Ubuntu versions as well as upcoming features that were undergoing development or discussion as it was written.

In this chapter, each alternate installation source is visited in turn, and the installation procedure explained.

3.1 The Ubuntu LTTng suite

The procedure is the same as in the *LTTng Quick Start Guide*, except that no repository need be added to the system’s software sources. The Ubuntu LTTng suite is also packaged a little differently than the other sources: it is broken down into a greater number of packages, although the ultimate contents are functionally the same. See Table 3 in Section 3.3.1 for the details.

The dependency diagram of the Ubuntu LTTng packages is shown in Figure 14 below. Because by default Synaptic treats recommendations as dependencies, one sees that requesting installation of either one of the two packages `lttng-modules-dkms` or `lttng-tools` is enough to obtain the entire LTTng suite, with the natural exception of the development packages (`lib*-dev`). The `lttngtop` package is offered by the LTTng PPA (see Topical Note A.8) and is a third all-installing Ubuntu LTTng package.

The Ubuntu LTTng suite consists of the following packages:

<code>babeltrace</code>	<code>libbabeltrace-ctf-dev</code>
<code>libbabeltrace-ctf0</code>	<code>libbabeltrace-dev</code>
<code>libbabeltrace0</code>	<code>liblttng-ctl-dev</code>
<code>liblttng-ctl0</code>	<code>liblttng-ust-dev</code>
<code>liblttng-ust0</code>	<code>liburcu-dev</code>
<code>liburcu1</code>	
<code>lttng-modules-dkms</code>	
<code>lttng-tools</code>	

The development packages (right-hand column) are unnecessary if you just want to use LTTng. They become necessary if you want to do some development, that is to say exploit the LTTng application programming interface (API). Figure 15 shows how the packages are organised from a functional point of view.

You can install the packages in any of three equivalent ways, as explained in the *LTTng Quick Start Guide*. These explanations are repeated here with some additional details.

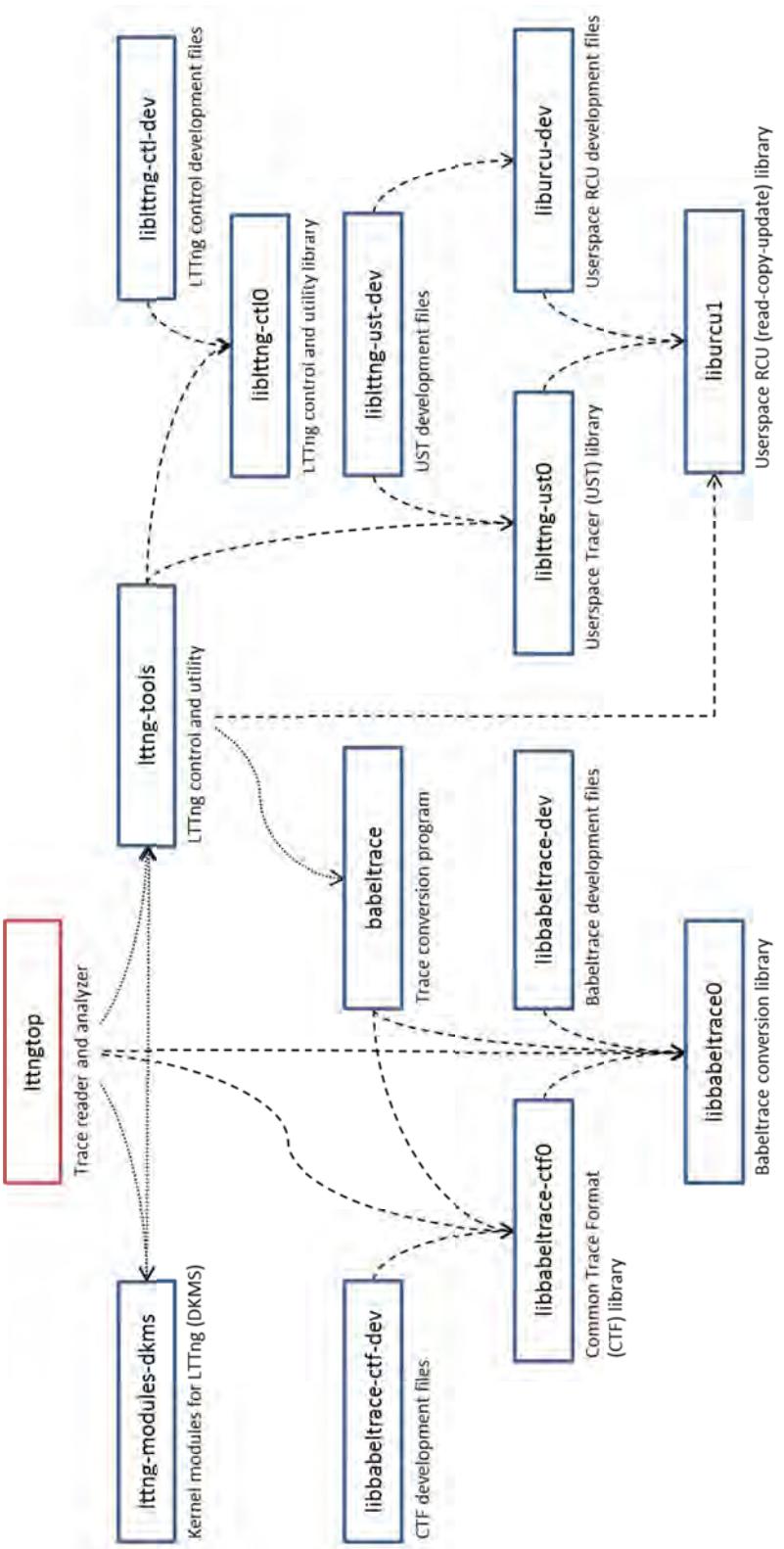
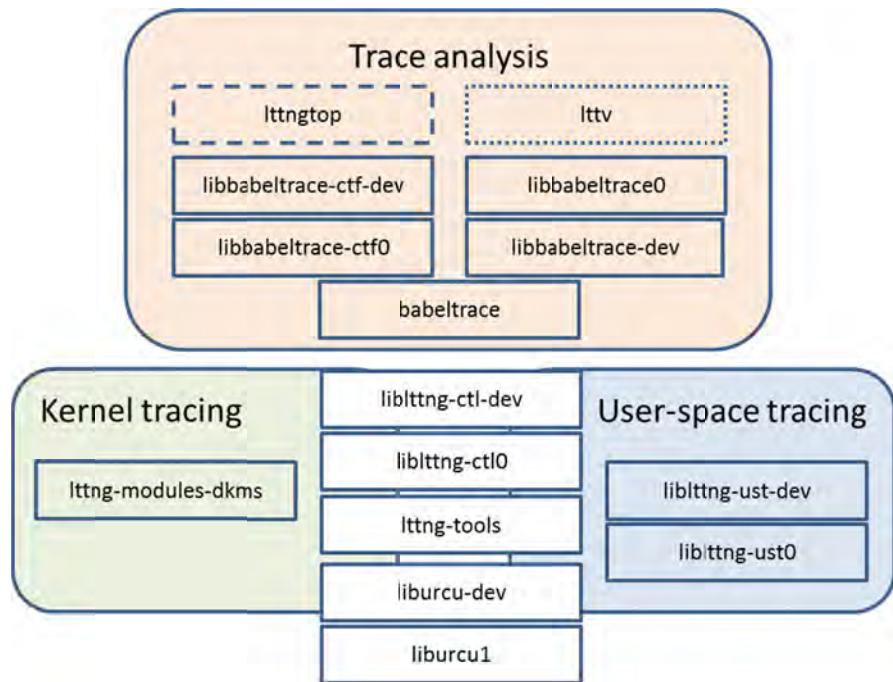


Figure 14: Dependencies and recommendations between the Ubuntu LTng packages. The former are shown as dashed arrows, the latter dotted. The `ltngtop` package (in red) is only available from the LTNg PPA.



*Figure 15: Functional break-down of the Ubuntu LTTng packages.
Compare with Figure 26.*

3.1.1 Using Ubuntu's Software Centre

Search for “LTTng”, choose “lttng-tools”, and click “Install” (see Figure 16 below).

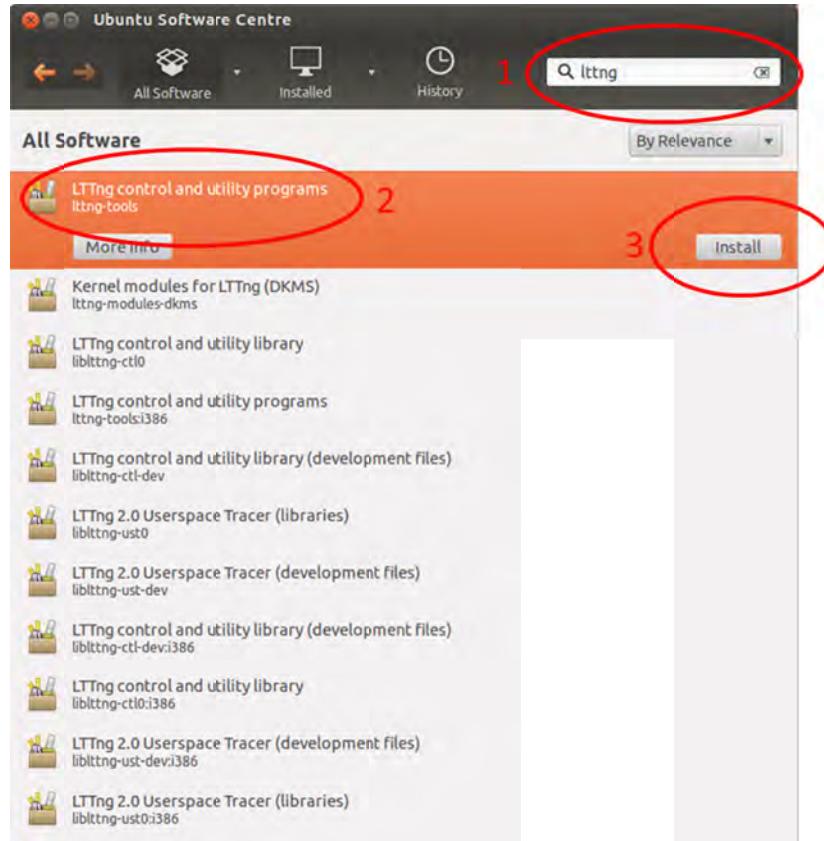


Figure 16: Ubuntu’s Software Centre and LTTng.
Search for “lttng” (1), choose “lttng-tools” (2) and click “Install” (3).

3.1.2 Using the Synaptic Package Manager

The Synaptic Package Manager is not installed by default; you can obtain it using Ubuntu’s Software Centre (search for “Synaptic”) or the command line (`sudo apt-get install synaptic`). It is recommended because it provides a high level of control over upgrades and installations, and also provides an easy way to find out what was installed by each package.

Using Synaptic, search for “lttng” using the filter, mark “`lttng-tools`” for installation (see Figure 17), acquiesce to the additional required changes (`babeltrace`, `libbabeltrace-ctf0`, etc.; see Figure 18), click “Apply” (see Figure 19) and confirm the installation (see Figure 20).

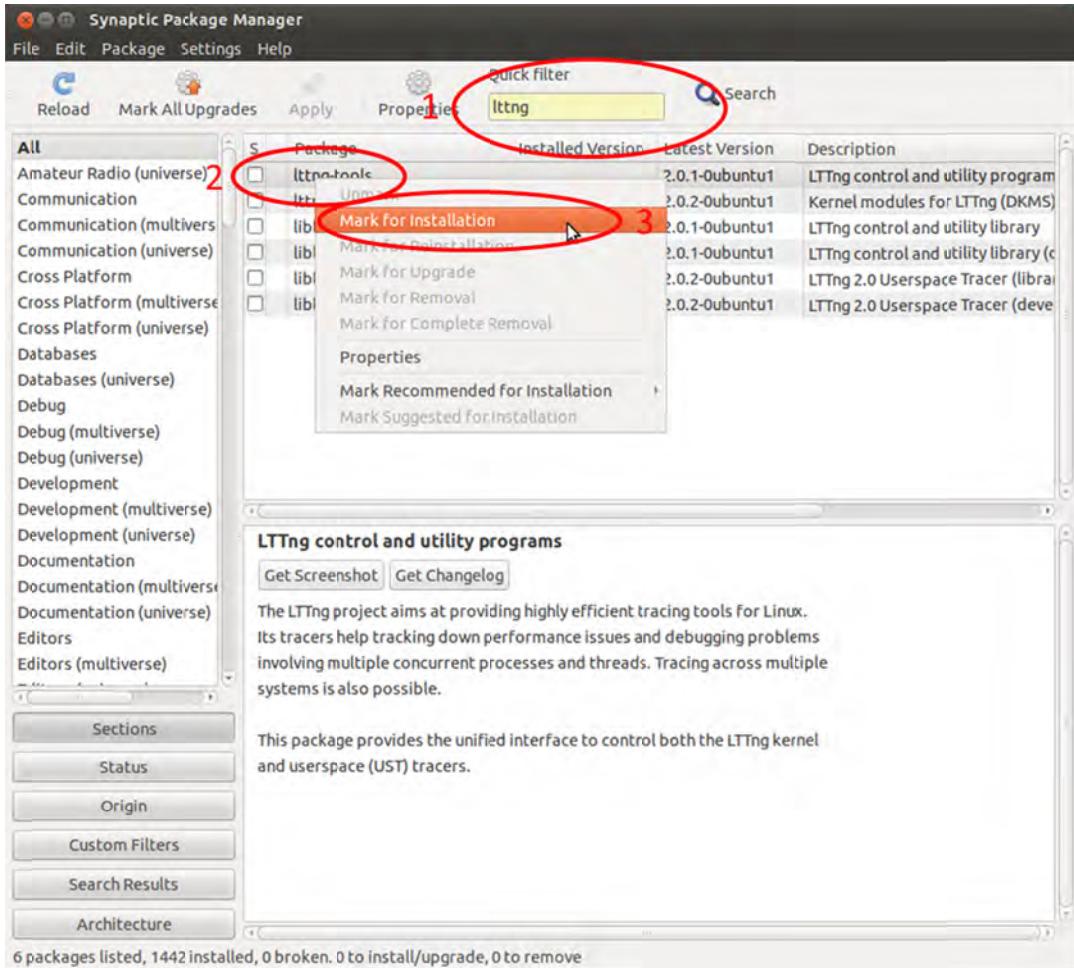


Figure 17: Ubuntu's Synaptic Package Manager and LTTng.

Search for “lttng” (1), choose “lttng-tools” (2) and mark it for installation (3). Here the contextual menu was used; you can also use the main menu’s “Package: Mark for Installation”.

Note that, by default, Synaptic treats package recommendations as dependencies. If you aim for a stripped-down installation, you may want to turn that off. The setting is Settings: Preferences: General: Marking Changes: Consider recommended packages as dependencies (and you must Reload to have the change applied).

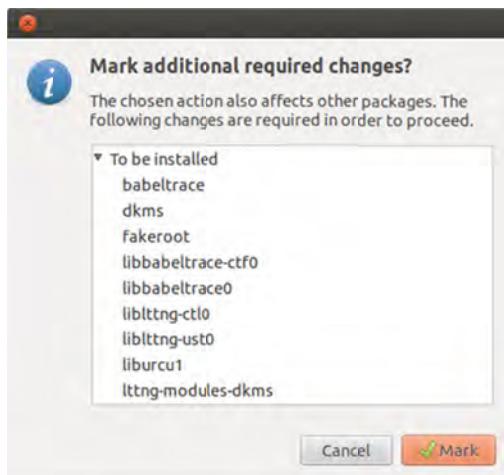


Figure 18: Synaptic requests the package's dependencies and recommendations.

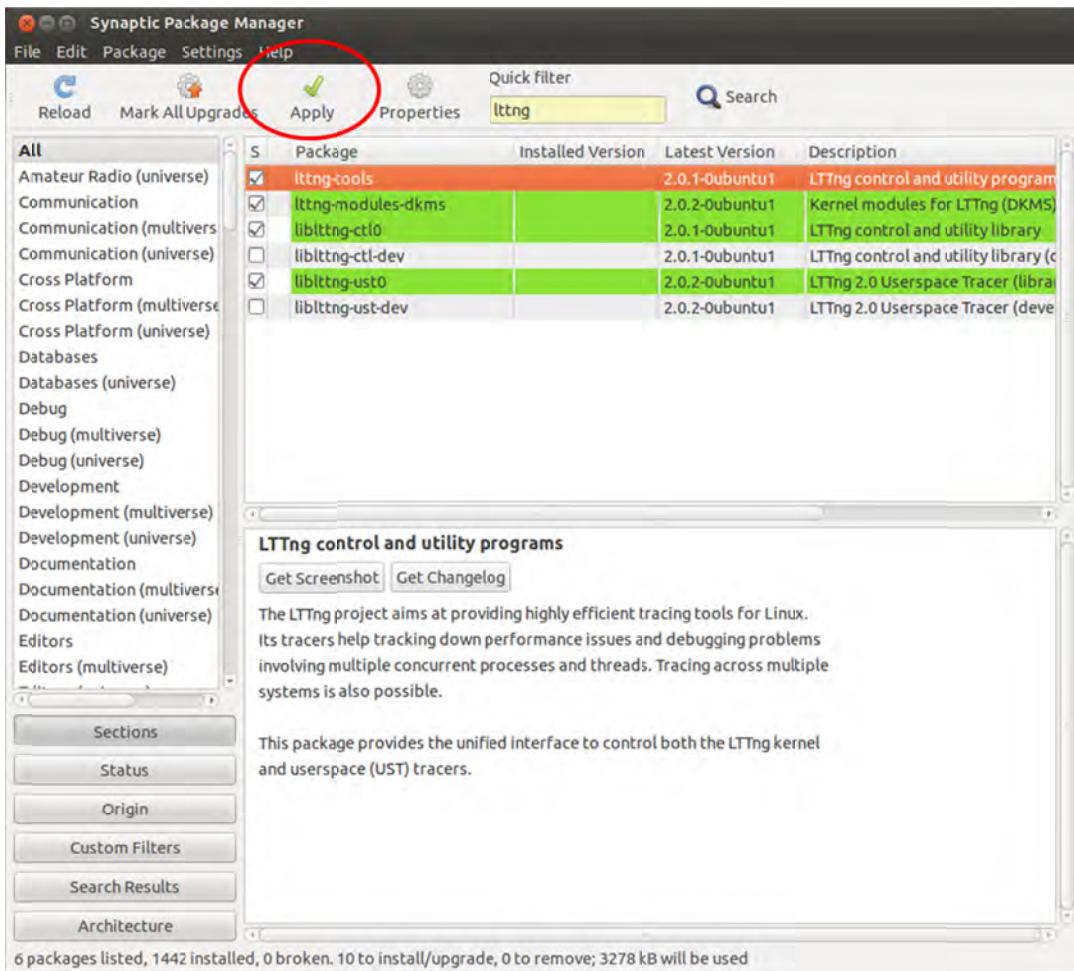


Figure 19: The Synaptic Package Manager is ready to apply the configuration changes. You can click the “Apply” button or choose the main menu’s “Edit: Apply Marked Changes”.

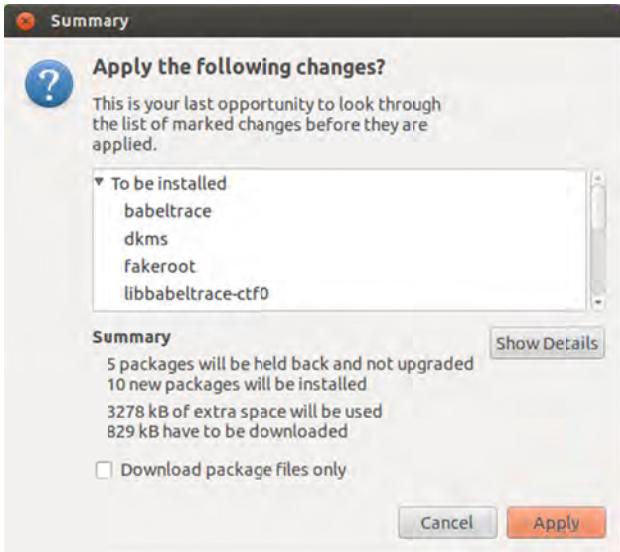


Figure 20: The Synaptic Package Manager requests final approval.

3.1.3 Using the command line

Run:

```
$ sudo apt-get install lttng-tools
```

3.1.4 Adding the `lttngtop` and `lttv` packages to the Ubuntu packages

The `lttngtop` and `lttv` packages are not yet readily available in the Ubuntu repositories. See Table 3 in Section 3.3.1 to find out whence they can be obtained. For instance, `lttngtop` is available on the LTTng PPA (albeit dated 2012-May-23). To add the LTTng PPA to the list of repositories in order to install `lttngtop` through Synaptic, see Topical Note A.8.

3.2 The DRDC (LTTng 2.3.0) suite

This procedure is explained in the *LTTng Quick Start Guide*. The instructions for adding a repository to Synaptic are repeated in Topical Note A.2.2. You should also turn off Synaptic's "treat recommendations as dependencies" default setting (see the end of Section 3.1.2). The `babeltrace`, `lttngtop` and `lttv` packages can be installed or left out, independently of the remaining ones. In order to trace the kernel, install the `lttng-modules`; to trace user-space applications, install `lttng-ust`—you can install one or the other or both. Once that installation is completed, install `lttng-tools`. It is actually possible to install `lttng-tools` while neither `lttng-modules` nor `lttng-ust` are present, but `lttng` won't be of much use (kernel support is always present in `lttng-tools` but can't be used until `lttng-modules` is installed; user-space support is actively removed from `lttng-tools` if it is built without the presence of `lttng-ust`).

Because of the idiosyncrasies of Debian package management, simultaneous installation of `lttng-ust` and `lttng-tools` is not recommended (`lttng-tools` would be built before `lttng-ust` and you would need to re-install the former in order to recover user-space tracing support). A complete install is best done in three passes: on the first pass, install `lttng-modules`, `userspace-rcu` and `babeltrace`. On the second pass, install `lttng-ust`, `lttngtop` and `lttv`. On the last pass, install `lttng-tools`. Since late 2013, with the introduction of live trace access for the trace analysis tools, the installation of `lttngtop` must be pushed back to a fourth pass (because it now depends on `lttng-tools`).

The version of `lttngtop` found in the LTTng PPA is 1.0~pre-0 and will thus hide the DVD-ROM’s `lttngtop`, version 0.2-10. But the LTTng PPA’s `lttngtop` release date of 2012-05-22 03:49 indicates it actually matches the 0.2-3 release found on git.lttng.org. To install the DVD-ROM’s `lttngtop` while you have the LTTng PPA in the software sources, simply go to the Software Sources dialog (see Topical Note A.8), uncheck the PPA, and refresh Synaptic. Once the DVD-ROM’s `lttngtop` is installed, you can turn the PPA software source back on (and refresh Synaptic again). Another solution is to select the `lttngtop-1.0~pre-0` package and then pick Synaptic’s `Package: Force Version...` menu. A window will appear with a drop-down list of all known versions of the selected package, from which you can pick `lttngtop-0.2-10` for installation. Clicking the `Force Version` button brings forth the usual “additional required changes” confirmation dialog, and the package will then appear selected “for downgrade” in the Synaptic listing.

In all cases, the sources will be installed in `/usr/src` and the binaries (and kernel modules) built and installed. Each step of the builds (`bootstrap`, `config`, `make` and `install`) is logged. You can observe the process by asking Synaptic to display the details (see Figure 21 below). The DVD-ROM’s packages simply automate the steps detailed in Section 3.3.3.

The `lttng-ust` and `lttng-tools` packages automatically detect the system configuration. Thus, if you have installed `swig2.0` the Python bindings for `lttng-tools` will be generated. Likewise, if you have installed `systemtap-dev` and/or a Java development kit (JDK), `lttng-ust` will support them.

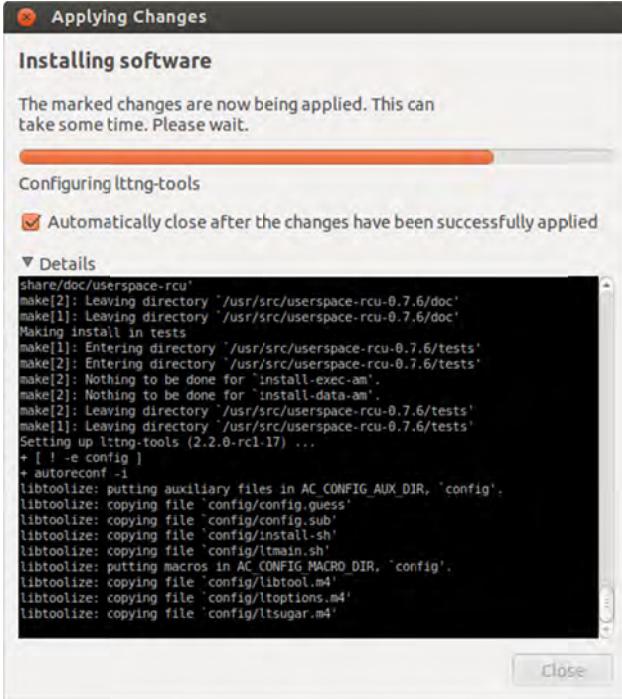


Figure 21: The installation details of the `lttnng-tools` package.

Whenever you change your configuration, re-installation of some of the `lttnng` packages may be necessary. For instance, `lttnng-modules` should be re-installed if the kernel configuration changes (not all kernel changes warrant a re-installation); `lttnng-tools` needs to be re-installed if `lttnng-ust` or `swig2.0` are added (the former in order to recover user-space tracing support, the latter in order to obtain Python bindings); if you install a JDK or SystemTap, `lttnng-ust` will need to be re-installed to recover Java/JNI (Java Native Interface) or SystemTap support (respectively).

3.3 The LTTng source code suite

3.3.1 Fetching the LTTng tarballs

Most of the source code tarballs are found on <http://git.lttng.org>. The `babeltrace` tarball is stored in a different location, <http://git.efficios.com>. The `babeltrace` tarball is not necessary in order to *generate* traces, but comes into play when viewer applications need to *read* the traces (see Figure 26 in Section 3.3.3).

The equivalencies between the various packages and tarballs are as shown in Table 3 (**drdc** is the DVD-ROM’s software repository).

Table 3: Package and tarball equivalencies by source.

Ubuntu 12.04 LTS	git. lttng. org	git. efficios. com	drdc [†]	LTTng PPA
babeltrace libbabeltrace0 libbabeltrace-dev libbabeltrace-ctf0 libbabeltrace-ctf-dev		babeltrace	babeltrace	babeltrace
lttng-tools liblttng-ctl0 liblttng-ctl-dev	lttng-tools		lttng-tools	lttng-tools
liblttng-ust0 liblttng-ust-dev	lttng-ust		lttng-ust	lttng-ust
liburcu1 liburcu-dev	userspace-rcu		userspace-rcu	liburcu
lttng-modules-dkms	lttng-modules		lttng-modules	lttng-modules
	lttngtop		lttngtop	lttngtop
	lttv		lttv	

[†]This represents the DVD-ROM’s software repository.

When using the git Web interface (Figure 22) of each project (e.g. when browsing <http://git.lttng.org/?p=lttng-tools.git;a=summary>, for instance), you land on the `master` branch page by default (Figure 23). From there you can pick another branch head from the menu at the page’s bottom (e.g. `stable-2.1` instead of `master`, Figure 24), or you can pick the most recent commit or any specific tag from the middle menu (e.g. `v2.1.1`, Figure 25). Download the snapshot of your choice. The snapshot is a `.tar.gz` archive (a tarball), which you can store in some convenient location (e.g. `~/Documents/lttng/`) before extracting its contents to a working folder.

Ideally, the major version numbers of the `lttng-modules`, `lttng-tools`, and `lttng-ust` tarballs should match (e.g. “2.3.0”), while `userspace-rcu` (a.k.a. `liburcu`), `babeltrace`, `lttngtop` and `lttv` have their own version numbering tracks (i.e. “0.8.0”, “1.1.1”, “0.2” and “0.12.38”, respectively).

projects /					
Search:					
Project	Description	Owner	Last Change		
benchmarks.git	Performance benchmarks for...	Julien Desfossez	2 years ago	summary	shortlog log tree
linux-2.6-ltng.git	LTTng 0.x enhanced Linux 2...	Mathieu Desnoyers	22 months ago	summary	shortlog log tree
ltt-control.git	LTTng 0.x kernel tracing contr...	Mathieu Desnoyers	23 months ago	summary	shortlog log tree
lttng-doc.git	LTTng documentation	Mathieu Desnoyers	2 years ago	summary	shortlog log tree
lttng-modules.git	LTTng 2.0/0.x modules	Mathieu Desnoyers	2 days ago	summary	shortlog log tree
lttng-tools.git	LTTng 2.0 tools and control...	David Goulet	4 hours ago	summary	shortlog log tree
lttng-ust.git	LTTng 2.0 Userspace Tracer	Mathieu Desnoyers	5 hours ago	summary	shortlog log tree
lttngtop.git	LTTngTop ncurse top like appli...	Julien Desfossez	5 weeks ago	summary	shortlog log tree
lttv.git	LTTng 0.x Viewer (LTTV)	Yannick Brosseau	2 days ago	summary	shortlog log tree
userspace-rcu.git	Userspace RCU (urcu) for Linux	Mathieu Desnoyers	4 weeks ago	summary	shortlog log tree
ust.git	UST 0.x - Userspace Tracer	Mathieu Desnoyers	15 months ago	summary	shortlog log tree

Figure 22: The git.ltng.org Web page, listing the available projects.

projects / lttng-tools.git / summary					
summary shortlog log commit commitdiff tree					commit ▾ <input type="checkbox"/> search : <input type="text"/> <input type="checkbox"/> re
<hr/>					
description LTTng 2.0 tools and control repository					
owner David Goulet					
last change Fri, 12 Apr 2013 15:13:42 +0000					
URL git://git.ltng.org/lttng-tools.git http://git.ltng.org/lttng-tools.git					
<hr/>					
shortlog					
4 hours ago	David Goulet	Fix: remove unused path variables from session obj	master	commit	commitdiff
5 hours ago	David Goulet	Fix: update lttn.1 man and enable-channel help with...	master	commit	commitdiff
5 hours ago	David Goulet	Fix: use channel per domain default values	master	commit	commitdiff
22 hours ago	David Goulet	Fix: typos in the code base	master	commit	commitdiff
23 hours ago	David Goulet	Fix: deny multiple event types with enable-event	master	commit	commitdiff
23 hours ago	David Goulet	Fix: deny the same port for data and control URL	master	commit	commitdiff
<hr/>					
tags					
2 weeks ago	v2.2.0-rc1	Version 2.2.0-rc1	tag	commit	shortlog
2 months ago	v2.1.1	Version 2.1.1	tag	commit	shortlog
3 months ago	v2.1.0	Version 2.1.0	tag	commit	shortlog
4 months ago	v2.1.0-rc9	Version 2.1.0-rc9	tag	commit	shortlog
4 months ago	v2.0.5	Version 2.0.5	tag	commit	shortlog
4 months ago	v2.1.0-rc8	Version 2.1.0-rc8	tag	commit	shortlog
<hr/>					
heads					
4 hours ago	master		shortlog	log	tree
2 weeks ago	stable-2.1		shortlog	log	tree
3 months ago	cygwin-2.0-experimental		shortlog	log	tree
4 months ago	stable-2.0		shortlog	log	tree
11 months ago	benchmark		shortlog	log	tree
<hr/>					
LTTng 2.0 tools and control repository					Atom RSS

Figure 23: A project summary Web page.
It lists the master branch commits (top), tags (middle), and heads (bottom).

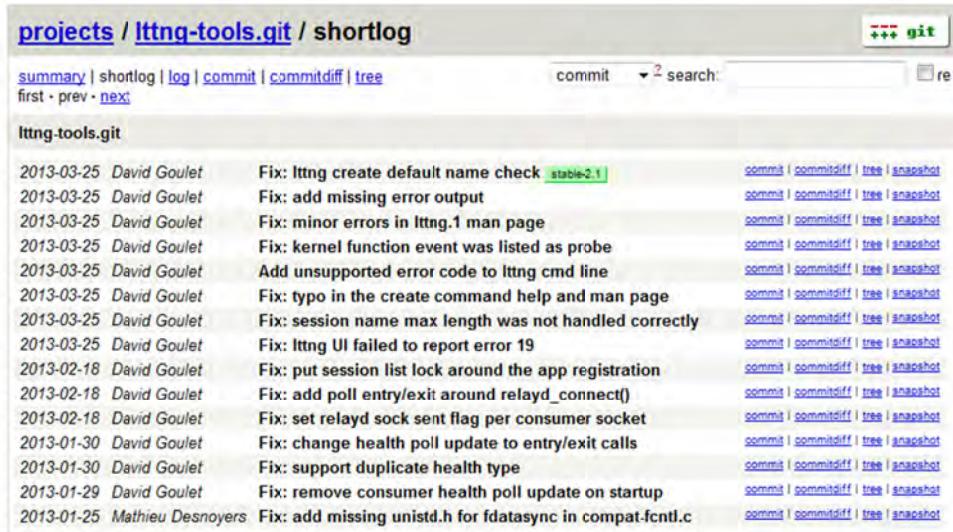


Figure 24: A head branch page, listing that branch's commits.



Figure 25: A tag commit page.

If you have git installed (`sudo apt-get install git` or “git” through Synaptic), you can clone the commit of your choice directly from the command line:

```
$ cd <path-to-working-folder>
$ git clone --branch <branch-name> git://git.lttng.org/<project>.git
```

Where for example `<path-to-working-folder>` could be `~/Documents/lttng`, `<branch-name>` could be `stable2.1`, and `<project>` could be `lttng-ust`. This would deploy the `stable2.1` branch to `~/Documents/lttng/lttng-ust`.

3.3.2 Preparing the system

It is probably best to remove any other LTTng packages or tarballs (such as Ubuntu’s `liblttng-*`, `lttng-modules-dkms`, `lttng-tools`, `liburcu*`, `babeltrace`, `libbabeltrace*`) if they were previously installed. Ensure that the various libraries identified in each of the downloaded tarballs’ `ReadMe` files are installed. As of this writing, this means (including some optional packages; packages included in Ubuntu’s baseline are omitted):

```
automake (this will also install autoconf and autotools-dev),  
bison,  
flex,  
libglib2.0-dev,  
libgtk2.0-dev,  
libncurses5-dev,  
libpango1.0-dev,  
libpopt-dev,  
libtinfo-dev,  
libtool,  
perl,  
python-dev,  
swig2.0,  
uuid-dev, as well as  
your kernel’s headers (e.g. linux-headers-generic).
```

You may also find `texinfo` useful (for some of the documentation installed by the tarballs). If you work with C++ applications, you will obviously need `g++` (see Section 4.3). Most missing packages are caught by `bootstrap` or `configure`, but there are occasional exceptions, such as `flex` and `bison`, which currently escape `lttng-tools`’s checks. Additional packages are needed if optional support for Java (Topical Note A.9) and SystemTap (Topical Note A.10) is desired. The `binutils-gold` package (the `gold` linker) was listed as required by `lttng-tools` up to and including the 2.1.1 release, but this requirement was relaxed with the 2.2.0 release (see Topical Note A.11).

3.3.3 Building and installing each LTTng tarball

You should build and install the tarballs in the following fetching order, extracted from the tarballs’ mutual dependencies as documented in their respective `README` files:

```
lttng-modules  
userspace-rcu  
lttng-ust  
lttng-tools  
babeltrace  
lttngtop  
lttv
```

This avoids cross-dependency problems. For example, `lttng-tools` depends on `lttng-ust` which depends in turn on `userspace-rcu`. The last three tarballs stand apart from the main LTTng suite (see Figure 26), but both `lttngtop` and `lttv` depend on `babeltrace`. Since late 2013, with the introduction of live trace access for the trace analysis tools, `lttngtop` now depends on `lttng-tools`.

When upgrading an existing LTTng installation, do make sure to stop and restart any `lttng` services (such as the session daemons, `lttng-sessiond`) once the upgrade is complete. See Section 2.1.1.1 for the service start/stop commands. Otherwise the running daemon(s) may run into trouble interpreting commands issued by an upgraded `lttng` executable. They may even crash or hang upon receiving upgraded user-space events, for instance. The DVD-ROM’s packages stop and restart the `lttng` services for you.

In specialised circumstances, you may want to install a pure kernel or pure user-space solution. That is to say, you can install LTTng in such a way as to support kernel tracing only, disallowing any user-space tracing—or the opposite: support user-space tracing only, disallowing any kernel tracing. Kernel tracing requires only `lttng-modules`, while user-space tracing requires only `lttng-ust`. The build of `lttng-tools` adapts itself to the presence or absence of either of these. Despite its name, `userspace-rcu` is actually required by `lttng-tools` even in the kernel-only case. This is because the read-copy-update (RCU) mechanism it implements is used in both domains. `Babeltrace`, `lttngtop` and `lttv`, dedicated to trace analysis, don’t require any LTTng installation at all. Both `lttngtop` and `lttv`, however, depend on `babeltrace`. These relationships are summarised by Figure 26. Note that with the introduction of live tracing, `lttngtop` became dependent on `lttng-tools` (`babeltrace` remained independent but gained a new entry format, ‘`lttng-live`’).

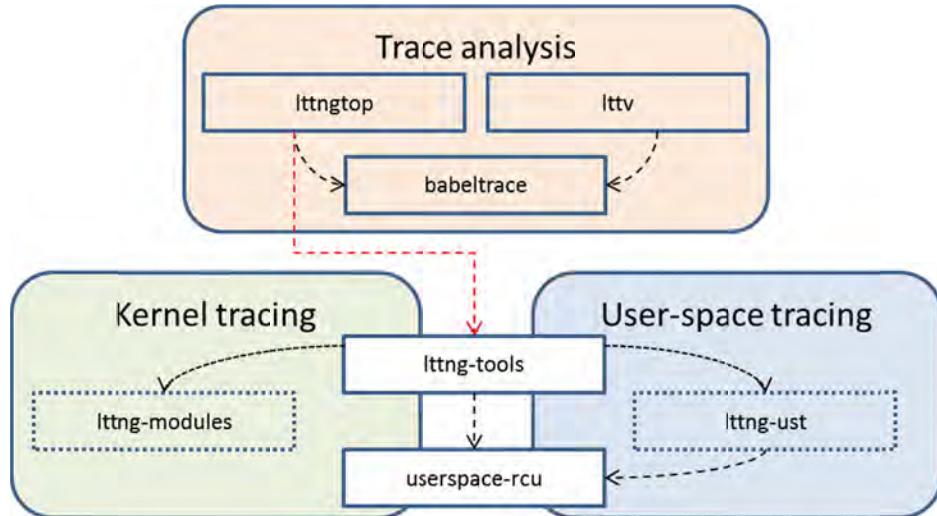


Figure 26: Functional break-down of the LTTng packages.

Kernel and/or user-space tracing is controlled by `lttng-tools` while `userspace-rcu` supplies read-copy-update buffers to both domains. Dependencies are indicated by dashed arrows; `lttng-tools` has “weak” dependencies (dotted arrows) to `lttng-modules` and `lttng-ust` (it adapts itself to their absence or presence). With the addition of live tracing, `lttngtop` becomes dependent on `lttng-tools`.

3.3.3.1 lttnng-modules

The `lttnng-modules` tarball installation procedure, as described in its `README` file, differs slightly from the remaining ones. Open a console in its folder and run the following commands, inspecting the corresponding log after each command to make sure no errors occurred:

```
$ make &> make.log  
$ sudo make modules_install &> install.log  
$ sudo depmod -a
```

Note: Unlike the Ubuntu `lttnng-modules-dkms` package, `lttnng-modules` does not re-build the LTTng kernel modules every time the kernel is changed. You will need to re-build and re-install the `lttnng-modules` whenever you install a new kernel. Registering the `lttnng-modules` with the `dkms` service is a delicate matter because of the numerous kernel configuration dependencies involved, and is beyond the scope of this document.

You can compare your kernel's actual configuration (found at `/boot/config-$(uname -r)`) with the required options mentioned in the `lttnng-modules` `README`. In particular, make sure your kernel has the following options specified (most kernels will):

```
CONFIG_HIGH_RES_TIMERS  
CONFIG_KALLSYMS  
CONFIG_MODULES  
CONFIG_TRACEPOINTS
```

You can check for each of these using a command line like this one:

```
$ cat /boot/config-$(uname -r) | grep CONFIG_HIGH_RES_TIMERS
```

If you do not intend to trace the kernel at all, the only kernel option that is truly required is `CONFIG_HIGH_RES_TIMERS`.

Some of the features of LTTng (e.g. system call tracing, B.4.10, context performance counters, B.4.1, etc.) are dependent on other kernel options; in their absence, the feature simply becomes unavailable. See the `lttnng-modules` `README` for details.

To build for a kernel other than the current one, use:

```
$ KERNELDIR=<path_to_kernel> make &> make.log  
$ sudo KERNELDIR=<path_to_kernel> make modules_install &> install.log  
$ sudo depmod -a <kernel_version>
```

Where `<path_to_kernel>` is `/lib/modules/<kernel_version>/build`.

The `lttnng-modules` tarball has no `make uninstall` target. This is rarely a problem; if you nevertheless need to “uninstall” the LTTng kernel modules, you need only delete the `lttnng*.ko` files found in `/lib/modules/$(uname -r)/extra` and its subfolders (after shutting down any running LTTng services, of course). Don't forget to run `sudo depmod -a` afterward.

3.3.3.2 userspace-rcu

Open a console in the tarball folder and run the following commands, inspecting the corresponding log after each command to make sure no errors occurred:

```
$ ./bootstrap &> bootstrap.log  
$ ./configure &> configure.log  
$ make &> make.log  
$ sudo make install &> install.log  
$ sudo ldconfig
```

The last command (`sudo ldconfig`) is probably not strictly required, since `make install` already invokes it partway through. Its invocation is purely precautionary.

The `configure` command recognises a variety of options (e.g. the `disable-smp-support` option will disable symmetric multi-processor synchronisation primitives, something you may want to do when the target system is uniprocessor); you can get a detailed list by running:

```
$ ./configure --help
```

Note that the `./configure` command is only available *after* running the `./bootstrap` command.

As mentioned in the `README` of `userspace-rcu`, you can also force `configure` to specify a particular build target by preceding (or following) it with a `CFLAGS` specification:

```
$ CFLAGS="-m32 -g -O2" ./configure will force a 32-bit build  
$ ./configure CFLAGS="-m32 -g -O2" will force a 32-bit build  
$ CFLAGS="-m64 -g -O2" ./configure will force a 64-bit build  
$ ./configure CFLAGS="-m64 -g -O2" will force a 64-bit build
```

Other possibilities are outlined in the `README`.

The installation can be tested using the `userspace-rcu/tests/runtests.sh` script. The script runs each of the test applications using the same parameters (e.g. `'./runtests.sh 2 2 10'`). You can also run the tests as root (e.g. `'sudo ./runtests.sh 2 2 10'`): you'll notice a marked slow-down due to kernel to user-space transitions.

The tarball can be uninstalled using `sudo make uninstall`.

3.3.3.3 lttng-ust

The procedure is the same as for `userspace-rcu`:

```
$ ./bootstrap &> bootstrap.log  
$ ./configure &> configure.log  
$ make &> make.log  
$ sudo make install &> install.log  
$ sudo ldconfig
```

The `./configure` line will need to be modified in some cases. First off, if you are using the alternate gold linker (the `binutils-gold` package), you may need to prefix (or suffix) it with `LDFLAGS=-L/usr/local/lib` so it can find the `userspace-rcu` libraries `lttng-ust` depends on (see Topical Note A.11). Finally, if you want `lttng-ust` to provide Java/JNI and/or SystemTap support, options must be passed to the `configure` command (see Topical Notes A.9 and A.10, respectively).

The procedure could thus be instead:

```
$ ./bootstrap &> bootstrap.log
$ LDFLAGS=-L/usr/local/lib ./configure \
  --with-java-jdk=/usr/lib/jvm/<jdk> \
  --with-jni-interface --with-sdt &> configure.log
$ make &> make.log
$ sudo make install &> install.log
$ sudo ldconfig
```

The `with-sdt` option turns on SystemTap support (the `systemtap-sdt-dev` package must already be installed, while the `systemtap` package may be installed later). See Topical Note A.10. The `with-java-jdk` and `with-jni-interface` options turn on Java and JNI support, respectively. The first option's argument, `/usr/lib/jvm/<jdk>`, is the installed JDK's path, such as for instance `/usr/lib/jvm/java-7-openjdk-amd64` (for the `openjdk-7-jdk` JDK). See Topical Note A.9.

To get a detailed list of the other `configure` command options, run (after `./bootstrap` has been run):

```
$ ./configure --help
```

Other options are detailed in the tarball's `README`. See Topical Note A.12 to see how to set compilation flags when instructed to do so by the `README`.

The installation can be tested using the `run.sh` script from the `lttng-ust/tests` folder (e.g. `'./run.sh unit_tests'`). Another test is running `make` from the `lttng-ust/doc/examples/easy-ust` folder. The `samplestatic` application this creates can be used to generate sample user-space events, or as an example of how to instrument an application (see Section 4.1.2).

The tarball can be uninstalled using `sudo make uninstall`.

3.3.3.4 lttnng-tools

The procedure is the same as for `lttng-ust`. Assuming the Python LTTng control module is desirable, the `configure` command is given an extra flag:

```
$ ./bootstrap &> bootstrap.log
$ ./configure --enable-python-bindings &> configure.log
$ make &> make.log
$ sudo make install &> install.log
$ sudo ldconfig
```

Like `lttng-ust`, `lttnng-tools` may need its `./configure` command prefixed (or suffixed) with `LDFLAGS=-L/usr/local/lib`.

To get a detailed list of the other `configure` command options (e.g. the `disable-lttnng-ust` option to build `lttnng-tools` for a kernel-only installation), run (after `./bootstrap` has been run):

```
$ ./configure --help
```

Note: The `lttnng-tools` packages (from Ubuntu and the DVD-ROM) configure your system in addition to simply installing the `lttnng` binaries. In particular, they set up the Upstart `lttnng-sessiond` daemon, create the tracing group, and set up `lttnng` command-line completion to make your life easier (see Topical Note A.1). If you want to do kernel tracing, you should join your user account to the tracing group. See Topical Note A.4 for details.

The installation can be tested using the `run.sh` script from the `lttnng-tools/tests` folder (e.g. ‘`sudo ./run.sh unit_tests`’). The tests will fail if you run them as a normal user but aren’t the owner of the build folder (this is the case when `lttnng-tools` was installed from the DVD-ROM; you can fix this with the ‘`sudo chown <user>:<group> -hR /usr/src/<folder>/`’ command on each of the build folders).

The tarball can be uninstalled using `sudo make uninstall`.

3.3.3.5 babeltrace

```
$ ./bootstrap &> bootstrap.log
$ ./configure &> configure.log
$ make &> make.log
$ sudo make install &> install.log
$ sudo ldconfig
```

The `babeltrace` tarball is not necessary to generate traces: it is used to read them.

To get a detailed list of the other `configure` command options, run (after `./bootstrap` has been run):

```
$ ./configure --help
```

The installation can be tested using the `runall.sh` script from the `babeltrace/tests` folder (e.g. ‘`sudo ./runall.sh`’). As in the `lttng-tools` case, the tests will fail if you run them as a normal user but aren’t the owner of the build folder.

The tarball can be uninstalled using `sudo make uninstall`.

3.3.3.6 lttngtop

`Lttngtop`, which is not published as a Ubuntu package yet, is designed to read and browse traces in order to display various statistics such as CPU (Central Processing Unit) usage, `perf` counters, and I/O (Input/Output) bandwidth per file and process. Although currently implemented only for recorded traces, a live tracing version is in development.

The procedure is the same as for `lttng-ust`:

```
$ ./bootstrap &> bootstrap.log
$ ./configure &> configure.log
$ make &> make.log
$ sudo make install &> install.log
$ sudo ldconfig
```

Like `lttng-ust` and `lttng-tools`, the `configure` step may need to be prefixed (or suffixed) with `LDFLAGS=-L/usr/local/lib`.

To get a detailed list of the other `configure` command options, run (after `./bootstrap` has been run):

```
$ ./configure --help
```

The tarball includes no test suite. The tarball can be uninstalled using `sudo make uninstall`.

3.3.3.7 lttv

`Lttv`, which is not published as a Ubuntu package yet, is a stand-alone trace viewer. It does not yet fully support CTF (Common Trace Format, see Section 2.4), which is the format of the traces produced by the LTTng 2.x kernel and user-space tracers. Work has been ongoing to complete this support since June 2012 or so. An alternative viewer is included in the set of Eclipse LTTng Tracing and Monitoring Framework (TMF) plug-ins (CTF traces are supported). The Eclipse TMF LTTng plug-ins include trace control capabilities, obviating the need for command-line `lttng`. As with `ltnngtop`, `lttv` live tracing capability is planned for some time in the future.

The procedure is the same as for `lttng-ust`:

```
$ ./bootstrap &> bootstrap.log
$ ./configure &> configure.log
$ make &> make.log
$ sudo make install &> install.log
$ sudo ldconfig
```

Like `lttng-ust` and `lttng-tools`, the `configure` step may need to be prefixed (or suffixed) with `LDFLAGS=-L/usr/local/lib`. Two important `configure` options are `with-trace-sync` and `with-jni-interface`. The first adds trace synchronisation calculation capability, while the second provides Java/JNI support (see Topical Note A.9).

To get a detailed list of the other `configure` command options, run (after `./bootstrap` has been run):

```
$ ./configure --help
```

The tarball includes no test suite. The tarball can be uninstalled using `sudo make uninstall`.

To run `lttv`, you can invoke:

```
$ lttv-gui
```

And then add a trace, or start `lttv` with the trace:

```
$ lttv-gui --trace <trace_path>
```

Where `<trace_path>` is a leaf trace folder (e.g. `lttng-traces/kertrace-20130418-150430/kernel` or `lttng-traces/usttrace-20130418-150436/ust/pid/sample-8617-20130418-150505`).

3.3.4 Testing for the kernel modules and manually loading them

You can test whether the LTTng kernel modules (or which ones) are currently loaded with the “`lsmod | grep lttng`” command, which returns nothing when no module has been activated. To manually load a module, you can use `modprobe`:

```
$ sudo modprobe <module>
```

Where `<module>` is the title of a `.ko` file found in `/lib/modules/$(uname -r)` or one of its subfolders (the nine to forty-odd LTTng kernel modules live in the `/extra`, `/extra/lib` and `/extra/probes` subfolders). Each `modprobe` command loads a module and all its dependencies, but since there are relatively few cross-dependencies between the LTTng kernel modules (`lttng-tracer` and `lttng-lib-ring-buffer` are used by 36 and 6 other modules respectively, and `lttng-tracer` in turn uses 4 other modules; the complete dependency tree is thus no deeper than three levels, although with only one isolated module, `lttng-types`), this does not save much work. You can get a complete list of the LTTng kernel modules and their loading order from `lttng-tools/src/bin/lttng-sessiond/modprobe.c` (where `lttng-tools` is the location where the tarball was deployed). If you must do this often, writing a script is definitely in order.

Unloading a module proceeds essentially the same way:

```
$ sudo modprobe -r <module>
```

3.3.5 Registering the root session daemon with Upstart

In Unix/Linux, `init` is the very first process launched by the kernel (it will nearly always have process ID 1) and will be the last process to terminate upon system shut down. It is the direct or indirect ancestor of all other processes. Several implementations exist, of varying mutual compatibility, depending on the operating system family (System V, BSD, etc.). The two main ones are arguably `systemd` (used by Fedora, Mandriva, and openSUSE, among others) and Upstart (used by Red Hat Enterprise Linux, Google's Chrome OS, and Ubuntu, among others). See <https://help.ubuntu.com/community/UbuntuBootupHowto> for details on Ubuntu's Upstart.

One of the responsibilities of `init` is the launching of system *services*, most of which are daemons handling network requests, hardware activity, scheduled tasks, and a variety of other jobs. It is advantageous to register the `lttng-sessiond` daemon as a service: it will be started early during system boot, and will thus be immediately available whenever you need to run a tracing session. The `lttng-tools` package of the accompanying DVD-ROM's software repository automatically registers the `lttng-sessiond` daemon as a service.

If you build `lttng-tools` from a tarball, however (see Section 3.3.3.4), you will need to register the daemon yourself. Registration with Upstart is achieved by the creation of an Upstart configuration file in `/etc/init` (named `lttng-sessiond.conf` by the `lttng-tools` package), whose contents are (you will need `sudo` privileges in order to do this):

```
description "LTTng 2.3 central tracing registry session daemon"
author "Stéphane Graber <stgraber@ubuntu.com>"
start on local/filesystems
stop on runlevel [06]
respawn
exec lttng-sessiond
```

The Ubuntu `lttng-tools` package also adds an `/etc/init.d` shortcut to the `/usr/bin/lttng-sessiond` executable, but this is not strictly necessary: `/etc/init.d` was used by the old System V `init`, which Upstart supplants.

3.4 Controlling the kernel and user-space session daemons

In this section, the behaviour of the LTTng suite under various combinations of root and user-space session daemons is discussed. The scope of the local and root daemons is briefly discussed first.

Upon being started, an instrumented application automatically registers itself with the user’s `lttng-sessiond` daemon and the root session daemon. If the session daemons aren’t running at the time the application starts, it spawns a small sub-process that will wait for the session daemons in order to complete the registration at that time. The local and root daemons coincide in the special case of root applications (such as daemons). Events emitted by an application instance are handled by *both* daemons (the root daemon and the instance’s user-space daemon) and can thus be committed to the owning user’s traces and to the traces of all other users that have access to the root daemon—that is to say, all users who either belong to the `tracing` group or have `sudo` privileges. See Figure 28 and Figure 29 for examples.

Users can be grouped into three broad groups. The *sudoers* are those users who have `sudo` privileges, and the *tracers* are those users who are members of the `tracing` group. A sudoer can also be a tracer and vice-versa. Those users who are in neither class are *normals*.

- A normal can trace only instrumented applications running in his user space (i.e., those applications he invoked himself). This is because his `lttng` commands will be handled by his local session daemon.
- A tracer can trace the kernel and all instances of instrumented applications, regardless of which user-space they are running in (in fact, user-space events *cannot* be restricted to specific user-spaces). This is because his `lttng` commands will be handled by the root session daemon.
- A sudoer can do what a normal can (by issuing unprivileged commands to his user-space session daemon), and can also (by using `sudo` to issue commands to the root session daemon) do what a tracer can. He can even impersonate other normals (by using `sudo -u` to issue commands as those users). The only restriction is that normal traces and tracer traces must be in separate, parallel sessions; should they need combining, this will have to be done during trace analysis. Further, sudoers can examine and control all sessions handled by the root session daemon, whereas tracers do not see each other’s sessions (and cannot share control).

The exact behaviour of `lttng` and `sudo -H lttng` commands depends on the user’s tracing group membership (see Topical Note A.4) and the set of currently running `lttng-sessiond` daemons:

- ◆ If neither `lttng-sessiond` daemons are running, or both `lttng-sessiond` daemons are running, or just the local `lttng-sessiond` daemon is running:
 - Regardless of the user’s tracing group membership:
 - ◆ An `lttng` command will be handled by the local `lttng-sessiond` daemon, spawning it if necessary.
 - ◆ A `sudo -H lttng` command will be handled by the root `lttng-sessiond` daemon, spawning it if necessary.

- ◆ If only the root `lttng-sessiond` daemon is running:
 - If the user *does not* belong to the `tracing` group:
 - ◆ An `lttng` command will spawn a local `lttng-sessiond` daemon.
 - ◆ A `sudo -H lttng` command will be handled by the root `lttng-sessiond` daemon.
 - If the user *does* belong to the `tracing` group:
 - ◆ An `lttng` or `sudo lttng` command will be handled by the root `lttng-sessiond` daemon. (See the Note below)
 - ◆ A `sudo -H lttng` command will be handled by the root `lttng-sessiond` daemon. (See the Note below)

Any `lttng-sessiond` daemons running in other user-spaces are ignored. If the root `lttng-sessiond` daemon was launched with the `group` option (see Section C.3), then the user's group membership is checked against the specified group name instead of `tracing`.

Note: It is very important to use `sudo -H lttng` commands rather than `sudo lttng` commands (which obey the same dispatching rules as the `sudo -H lttng` commands). This is because `lttng` keeps track of the “current tracing session” through a small hidden file, `~/.lttngrc`. Using `sudo -H` ensures that the root’s home is `/root` and that there will be no contention for the `~/.lttngrc` file. If just `sudo` were used, the root and local `lttng-sessiond` daemons would overwrite each other’s current session or, if the root daemon is the one to create the `.lttngrc` file, root ownership would prevent the local daemon from accessing the file, giving rise to unexpected error messages. This is of course not a problem if `sudo` commands and non-`sudo` commands aren’t interleaved from any given user account.

In the special case where just the root `lttng-sessiond` daemon is running and the user belongs to the `tracing` group, the `lttng` and `sudo -H lttng` commands will have different “current sessions”. Another key difference between these two cases is that the `lttng` traces are stored in the user’s `~/lttng-traces` folder and *he gets ownership* (even though they are written by a root `lttng-consumerd` daemon), whereas the `sudo lttng` traces (also stored in the user’s `~/lttng-traces` folder) and `sudo -H lttng` traces (stored in the `/root/lttng-traces` folder) remain owned by root, which makes them more difficult to access. Finally, a subtlety of `lttng` compared to `sudo lttng` in this special case is that `lttng list` will only show your own sessions, whereas `sudo lttng list` will show all sessions managed by the root daemon, including in particular the other `tracing` group user sessions. Any session mentioned by `lttng list` or `sudo lttng list` can be manipulated.

Impersonating other users: If you use `sudo -u <username>` to issue `lttng` commands on behalf of another user, you must also use the `-H` option (i.e. `sudo -H -u <username> lttng`). Otherwise not only are you going to have `.lttngrc` file contention, but the commands will likely fail because the other user may not have permission to create files in your home folder. You may need to impersonate users in this way if you must segregate user-space events on a process owning user ID basis. It is also possible to achieve this segregation *a posteriori* during trace analysis, or *in vivo* (so to speak) by adding filters (see Section B.4.10) that use the `getuid()` or `geteuid()` system calls (but this would incur a comparative performance cost). It is a matter of choice between storage space, performance, and convenience.

The `su` command: Another way to send commands to the root session daemon is to open a root shell with the `su` command. The default configuration under Ubuntu is to lock the root user: one cannot log in as root, so `su` cannot be used (`su` is also out of the question when one does not know the root password, obviously). A root shell may nevertheless be opened using the `sudo -s` command, but since staying in such a shell is somewhat risky, this document instead recommends using `sudo -H` on a command-by-command basis. The `-H` option may be omitted if the `sudo` security policy sets the `HOME` environment variable of the elevated command to “`root`” (this is *not* the default under an Ubuntu configuration).

In order to avoid confusion, you should define your tracing needs at the outset and stick to an established tracing command protocol. If you only need to trace your own user-space, you should not join the `tracing` group, and you should not use `sudo lttng`. In all other cases, whether you need to trace the kernel or other users’ user-spaces, it is probably best to join the `tracing` group, make sure Upstart keeps the `lttng-sessiond` daemon running, and use only `lttng` commands.

The `tracing` group: It is important to realise that in order for `tracing` group membership to be meaningful, the root session daemon *must* be running and the local session daemon *must not* be running. When joining or quitting the `tracing` group, do recall that group membership is applied during user login, so you must log off and back on to complete the change. The `lttng-sessiond` command has a `group` option (see Section C.3), so you can use some other name for the `tracing` group if you wish.

You may not run several `lttng-sessiond` root daemons (with different `group` options, for instance). Note also that there is currently no way to ask the root `lttng-sessiond` daemon directly what its current group setting is; to find out, issue the following command:

```
$ ls -dg /var/run/lttng | cut -f3 -d\
```

The command’s trailing space (shown escaped in the line above) is important. The command returns the tracing group name, or an empty string if the root daemon isn’t running.

3.4.1 Comparison between session daemon control modes

Figure 27, below, shows how control and data flow using `sudo -H ltng` commands. Each user shell is sending commands to (and receiving feedback from) the same unique `ltng-sessiond` daemon. Up to three `ltng-consumerd` daemons run as root, one handling the kernel events (some of which may be triggered by system calls issued by the traced applications), the other two handling the 32- and 64-bit user-space events issued by the traced applications. The traces are captured in root's `ltng-traces` folder (`/root/ltng-traces/`). Both users have access to all sessions handled by the root session daemon—they can control each other's sessions. There is one caveat in this operational model: the “current session” of the `ltng-sessiond` daemon is the same for both users, causing constant contention for the `/root/.ltngrc` file. It will be necessary to use mutually agreed-upon, distinct session names and to specify the target session of each `ltng` command, using either the explicit command argument (for the `create`, `list`, `start`, `stop`, `view`, and `destroy` commands) or the `session` command option (most other commands). The `set-session` command should not be used at all.

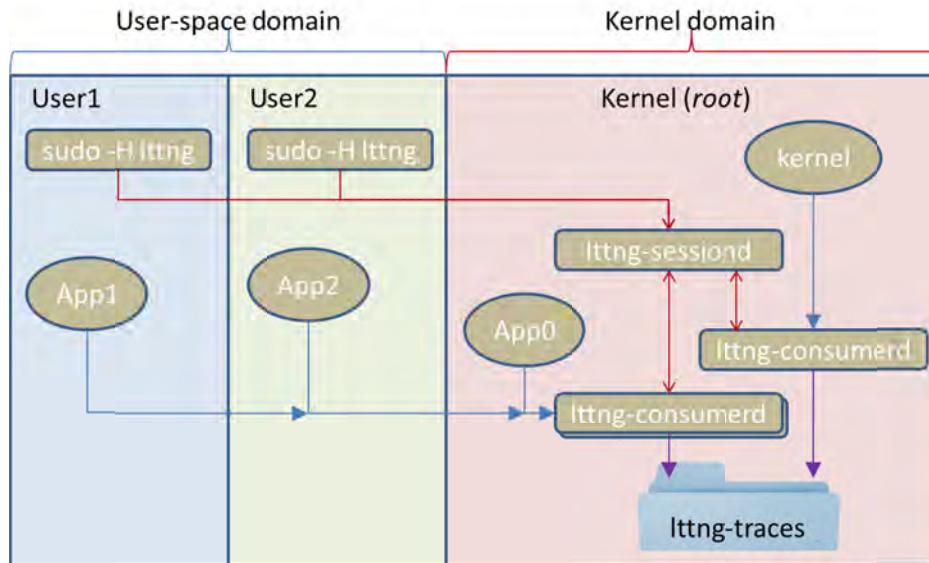


Figure 27: Tracing using sudo -H ltng.
 Red arrows: control flow; blue arrows: event data flow;
 purple arrows: event record flow.

Figure 28, below, shows how control and data flow using `ltng` (or `sudo ltng`) commands when one is a member of the tracing group. Each user shell is sending commands to (and receiving feedback from) the same unique `ltng-sessiond` daemon, as in the `sudo -H ltng` case. The same `ltng-consumerd` daemons running as root are still handling the kernel and user-space events. The first difference is that the events are being recorded in each user's respective `ltng-traces` folder, and the users are free to use the same session names without fear of contention. Each user captures all user-space events, as before. The second difference is that neither user has access to the sessions created by the other, even though they are both handled by the root session daemon—they can only control their own sessions.

Figure 29, finally, shows two LTTng-unprivileged users (meaning ones who do not belong to the root `lttng-sessiond` daemon's tracing group); the first uses `lttng` commands while the other uses `sudo lttng` commands (the second user is privileged in the sense that he belongs to the `sudo` group). The first user shell is sending commands to (and receiving feedback from) a local `lttng-sessiond` daemon. That daemon can only run local `lttng-consumerd` daemons and consequently can only trace user-space events issuing from its own user-space. The second user shell deals with the root `lttng-sessiond` daemon and can thus trace the kernel and all user-spaces. Because the second user is issuing LTTng commands as root, he can control the first user's sessions—whereas the first user cannot see nor control the second user's sessions.

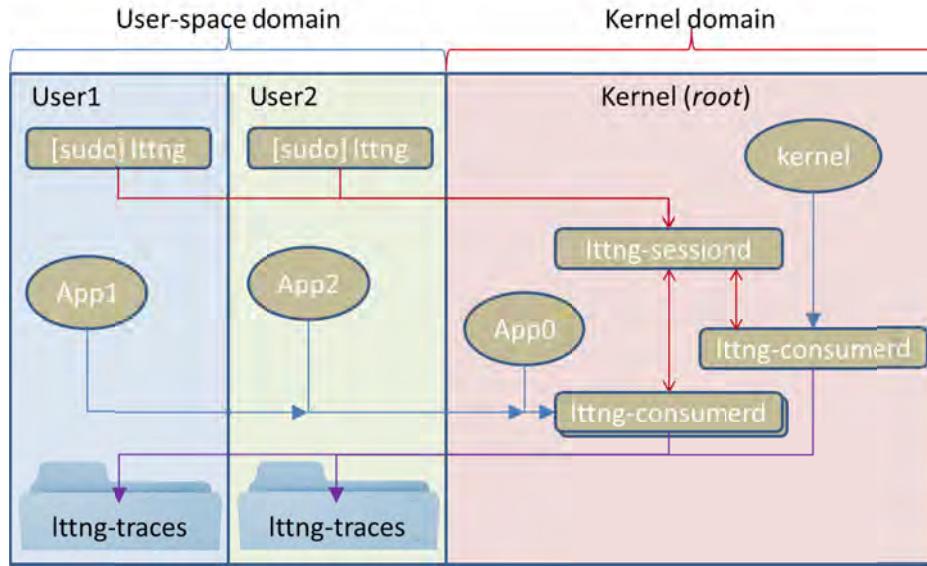


Figure 28: Tracing using lttng as a member of the tracing group.
Non-members can achieve the same using sudo lttng.

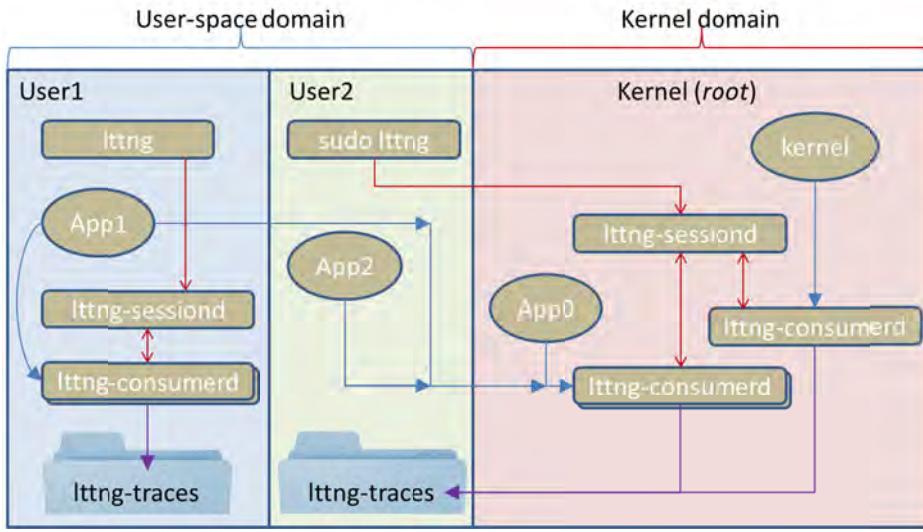


Figure 29: Tracing using lttngr or sudo lttngr while not a member of the tracing group.
Elevating the lttngr client with sudo switches session daemons.

To summarise:

- ◆ sudo lttngr commands are unrestricted (can trace kernel and all user-spaces); traces are stored in the local lttngr-traces folder, under root ownership; all root sessions are accessible.
- ◆ sudo -H lttngr commands are unrestricted; traces are stored in root's lttngr-traces folder, under root ownership; all root sessions are accessible.
- ◆ If the user belongs to the tracing group and only the root lttngr-sessiond daemon is running (if neither daemon is running, a root daemon will be spawned):
 - lttngr commands are unrestricted; traces are stored in the local lttngr-traces folder under user ownership; only the user's own root sessions are accessible.
- ◆ Otherwise (both lttngr-sessiond daemons are running, or just the local lttngr-sessiond daemon is running, or just the root lttngr-sessiond daemon is running but the user does not belong to the tracing group):
 - lttngr commands are restricted to the user's own user-space; traces are stored in the local lttngr-traces folder under user ownership; no root sessions are accessible.

4 Instrumentation of software in the user and kernel spaces

Now that LTTng has been installed and can be operated, how does one go about instrumenting applications and processes? The work flow is shown by Figure 30, below.

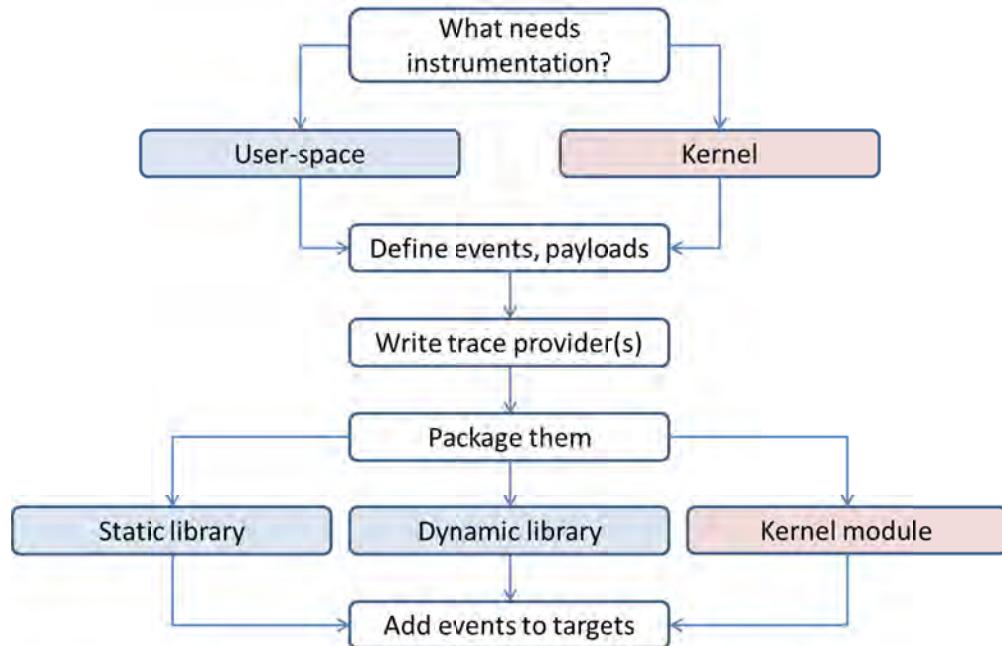


Figure 30: The tracepoint instrumentation work flow.

The process is always the same from a high-level perspective: decide what needs to be known and how best to get the system to tell what needs to be known, then design the tracepoints, package them and inject them where they need to be. It may very well be that the tracing needs are already appropriately covered by LTTng's built-in kernel tracepoints and other probes. In such a case, the work will reside entirely in the design and execution of tracing sessions. The rest of this chapter, by contrast, will focus on the remaining cases, where there is a need to add tracepoints to applications, kernel modules or the kernel itself.

4.1 Instrumenting a C application (user-space)

An example of instrumentation is given by the `doc/examples` build tree that comes with `lttng-ust` (currently obtainable from `git.lttng.org`, expected to be installed by the `liblttng-ust-dev` Ubuntu package sometime in the near future). The `lttng-ust` package in the DVD-ROM's software repository has been enriched with an additional `doc/examples/drdc` folder that holds several detailed examples of various application instrumentation options. These options will be discussed here.

Each application module that needs instrumentation simply adds `tracepoint()` instructions at the appropriate places and includes a *tracepoint header* in its prologue. The tracepoint header is also used to generate the *tracepoint provider* executable, which can then be linked in various ways to the target application. Adapting this to languages other than C or C++ is usually a simple matter of importing the `tracepoint()` C function—as long as they are compiled languages. The Java case, where a virtual machine is used, is discussed in Section 4.4.

There can be an arbitrary number of tracepoint providers within an application, but they must each have distinct provider names (duplicate provider names are not allowed). Each tracepoint provider may define several tracepoint events, each with its own name and arguments. All tracepoints are called using the `tracepoint` macro, whose first two arguments are the tracepoint provider name and the tracepoint event name. You cannot “overload” a tracepoint provider’s event like you can a method in Java: each tracepoint event supplied by a given provider has a single argument signature.

4.1.1 Polymorphous tracepoints

In user-space (and possibly in the kernel domain as well), it is possible to have different processes emit events, using different tracepoint providers, that have the same fully qualified names (same provider names and same event names) with different argument signatures (different payloads). One provider’s event payload could be a character string while the other provider’s event payload could be an integer, for instance. Such a situation should be avoided. By default, user-space events are captured on a per-user-ID basis and the process IDs are not in the event data (unless included in the event payloads or added by the `add-context` command). Currently, the `metadata` captures the event definition only once per fully qualified name, so `babeltrace` won’t be able to decode the homonymous events correctly: it may generate spurious event records or a fatal error partway through [21].

4.1.2 Adding LTTng probes in software applications

For demonstration purposes, a very simple application (see `/usr/src/lttng-ust-2.3.0/doc/examples/drdc/sample.c` after installing the DVD-ROM's `lttng-ust` package) is used:

```
#include <stdio.h> //For printf
#include <unistd.h>

int main(int argc, char **argv)
{
    int i = 0;
    char themessage[20]; //Can hold up to "Hello World 9999999\0"

    fprintf(stderr, "sample starting\n");
    for (i = 0; i < 10000; i++) {
        sprintf(themessage, "Hello World %u", i);
        usleep(1);
    }
    fprintf(stderr, "sample done\n");
    return 0;
}
```

This sample application simply sleeps 10,000 times for 1 millisecond each time, modifying an internal “Hello World” string every time. A tracepoint will be added to this application in order to trace the internal string. First, the source is modified like this:

```
#include <stdio.h> //For printf
#include <unistd.h>
#include "tp.h"

int main(int argc, char **argv)
{
    int i = 0;
    char themessage[20]; //Can hold up to "Hello World 9999999\0"

    fprintf(stderr, "sample starting\n");
    for (i = 0; i < 10000; i++) {
        sprintf(themessage, "Hello World %u", i);
        tracepoint(sample_component, event, themessage);
        usleep(1);
    }
    fprintf(stderr, "sample done\n");
    return 0;
}
```

The added lines are in red. Note that the first two tracepoint macro arguments (the tracepoint provider name and tracepoint event name) must be *literal* strings: they cannot be string variables. The remaining arguments are variables or constants, just as with regular functions. LTTng's UST currently supports between zero and ten arguments (inclusive); adding an eleventh argument is simply a matter of extending the `tracepoint.h` macro prototypes.

Below is the fragment of the Makefile used to make the instrumented executable. The lines highlighted in various colours are discussed afterward.

```
CC = gcc
LIBDL = -ldl # On Linux
LIBUST = -llttng-ust
#LIBS = $(LIBDL) $(LIBUST)
LOCAL_CPPFLAGS += -I.
LDFLAGS += -L/usr/local/lib
TP_DEFINE = -D TRACEPOINT_DEFINE

all: static_samples

static_samples: static
#####
#Trace provider statically included.
static: static.o tp.o
    @echo "~~~~~Linking sample $@"
    $(CC) -o sample_$@ $^ $(LDFLAGS) $(LIBDL) $(LIBUST)
    @echo "    Use './sample_$@' to run sample_$@"

static.o: sample.c tp.h
    @echo "~~~~~Compiling $@"
    $(CC) $(CPPFLAGS) $(LOCAL_CPPFLAGS) $(TP_DEFINE) -c -o $@ $<

tp.o: tp.c tp.h
    @echo "~~~~~Compiling $@"
    $(CC) $(CPPFLAGS) $(LOCAL_CPPFLAGS) -c -o $@ $<
```

The compilation instructions are straightforward. Note the `$(TP_DEFINE)` when compiling `static.o` (highlighted in blue). It is equivalent to including this line in the preamble of `sample.c`:

```
#define TRACEPOINT_DEFINE
```

Using a compiler flag to set `#define` values is more flexible. For instance, if `sample.c` were written like this:

```
#include <stdio.h> //For printf
#include <unistd.h>
#ifndef SAMPLE_TRACEPOINTS
#include "tp.h"
#endif

int main(int argc, char **argv)
{
    int i = 0;
    char themessage[20]; //Can hold up to "Hello World 9999999\0"

    fprintf(stderr, "sample starting\n");
    for (i = 0; i < 10000; i++) {
        sprintf(themessage, "Hello World %u", i);
#ifndef SAMPLE_TRACEPOINTS
        tracepoint(sample_component, event, themessage);
#endif
        usleep(1);
    }
    fprintf(stderr, "sample done\n");
    return 0;
}
```

Then the tracepoints can be turned on and off simply by flicking a `$ (USE_TRACEPOINTS)` compiler switch (makefile line: `USE_TRACEPOINTS = -D SAMPLE_TRACEPOINTS`). See Topical Note A.12 for further details. It would not be a good idea to use the `$ (TP_DEFINE)` compiler switch for this purpose, as explained in Section 4.1.2.1, below.

The addition of `/usr/local/lib` to `LDLIBS` (highlighted in green) is usually redundant; it is done here only in order to proof the makefile against the so-called GNU `gold` bug (see Topical Note A.11).

The linking line (highlighted in yellow) can undergo several variations (see Section 4.1.3). The dynamic linker (`libdl` on Linux, but `libc` on BSD) is necessary because `lttng` uses dynamic linking internally (more on this in later examples). The `liblttng-ust` library, finally, is what links all user-space tracepoints with the `lttng` modules.

Some extra care is needed when using `liblttng-ust` with daemon applications that call `fork()`, `clone()`, or BSD `rfork()` without a following `exec()` family system call. The library `liblttng-ust-fork.so` needs to be preloaded for the application (launch with e.g. `LD_PRELOAD=liblttng-ust-fork.so <appname>`).

4.1.2.1 The tracepoint provider code

The tracepoint provider supplies the `tracepoint` method and is defined by the usual pair of C files: `tp.c` and `tp.h`. However, all the work is accomplished by the header file, the C file being a mere generic excuse for compiling the header:

```
/*
 * Defining macro creates the code objects of the traceprobes.
 * Must be done only once per file (this #define controls
 * tracepoint.h and tracepoint-event.h).
 */
#define TRACEPOINT_CREATE_PROBES
/*
 * The header containing our TRACEPOINT_EVENTS.
 */
#include "tp.h"
```

You will rarely need to change the `tp.c` file. Section 4.1.2.2 shows how to use the `lttng-gen-tp` python script (see Annex C) to generate `tp.c` and `tp.h` from a `<tracepoint-provider-name>.tp` template. In any case, it is possible to do without the `tp.c` file entirely, by changing the `Makefile` lines from:

```
tp.o: tp.c tp.h
@echo "~~~~~Compiling $@"
$(CC) $(CPPFLAGS) $(LOCAL_CPPFLAGS) -c -o $@ $<
```

To:

```
tp.o: tp.h
@echo "~~~~~Compiling $@"
$(CC) $(CPPFLAGS) $(LOCAL_CPPFLAGS) -c \
-x c -D TRACEPOINT_CREATE_PROBES -o $@ $<
```

Each compilation unit (`.c` file) that wishes to call `tracepoint` must include the tracepoint header (`tp.h`) in its prologue. The `$(TP_DEFINE)` highlighted in blue in the Section 4.1.2 `Makefile` code above must be applied to an `#include`d tracepoint provider in precisely *one* of the application's modules⁴ (typically the main one); all it does is trigger the actual compilation of the tracepoint provider. If several application compilation units are compiled with `$(TP_DEFINE)`, you will get errors during their linking into a single module.

This is the reason why `$(USE_TRACEPOINTS)` and not `$(TP_DEFINE)` was mentioned at the end of the previous section: `tp.h` must be included in *each* source file where a `tracepoint()` call has been added, while `$(TP_DEFINE)` must be switched on in *just one* of the source files that make up the final application module.

⁴ The C language does not define the term “module”. Here it is used to mean each standalone executable or dynamic library. Each source (`.c`) file is a “compilation unit”, yielding an object (`.o`) file; multiple object files are linked together to form modules. An application’s memory space will typically contain one or more modules (generally the executable’s image and any dynamic libraries it loads).

There is one other `#define` instruction which affects the tracepoint provider compilation: `TRACEPOINT_PROBE_DYNAMIC_LINKAGE`. It accompanies the `TRACEPOINT_DEFINE` instruction (i.e. `$(TP_DEFINE)`) and is ignored if used elsewhere. It serves to mark the tracepoint provider for compilation into a dynamically linked library instead of a statically linked one, as shown in Section 4.1.3.3.

4.1.2.2 The tracepoint header

Trace providers are defined by a single template (`.tp5`) file which is surrounded by boilerplate code to become a pair of source (`.c`) and header (`.h`) files. Annex C details the `lttng-gen-tp` python script which can generate those from the template. This pair of files is in turn compiled into an object file (`.o`) for static inclusion in instrumented applications or, using a slightly different compilation configuration, into an object that can be bundled in a shared object (`.so`) for dynamic inclusion in instrumented applications.

The LTTng header `lttng/tracepoint.h` (the contents of `ust/tracepoint.h` are completely different) includes a number of naming guidelines (roundabout line 460) that should be followed when picking the tracepoint provider and event names.

The “provider:event” name cannot exceed 255 characters (see `LTTNG_UST_SYM_NAME_LEN` in `lttng-ust/include/lttng/ust-abi.h` and `lttng-tools/src/bin/lttng-sessiond/lttng-ust-abi.h`). If it does, the instrumented application will compile and run correctly, but LTTng will issue multiple “Warning: Truncating tracepoint name <provider_name>:<event_name> which exceeds size limits of 255 chars”. The events are nevertheless captured. However, if different events end up with the same truncated name (including the provider name part), serious problems will arise [22]: if their signatures differ, the instrumented application will almost certainly crash; if the signatures do not, the events will cross-trigger (each invocation will result in multiple event records).

Avoid using significant keywords as event names, as there could be unexpected naming conflicts. For instance, if your application uses complex numbers (and thus includes `/usr/include/complex.h`), the tracepoint invocations of events named `complex` won’t compile correctly (you will get this rather obscure error: ‘`__tracepoint_sample_component_complex`’ undeclared (first use in this function)). The tracepoint provider and event names must follow the rules for so-called C99 identifiers (see [23], section 6.4.2 and Annex D).

The tracepoint provider name is defined by the `#define TRACEPOINT_PROVIDER` instruction and is the namespace of the event names as reported by `lttng list` (see Section B.4.11).

⁵ In this case “`.tp`” stands for “template” and has nothing to do with “tracepoint”.

Each event in the tracepoint header is defined by a single `TRACEPOINT_EVENT` macro optionally followed by a `TRACEPOINT_LOGLEVEL` macro that specifies its logging level (by default, this is `TRACE_DEBUG_LINE`, just a shade more important than `syslog`'s last logging level, `LOG_DEBUG`). Another optional macro, `TRACEPOINT_MODEL_EMF_URI`, can be added to set the tracepoint's EMF (Eclipse Modeling Framework) model URI (Uniform Resource Identifier); see the `<EMF>` field in Section F.2.2.

4.1.2.3 The `TRACEPOINT_EVENT` macro

```
TRACEPOINT_EVENT(<provider name>, <event name>, <arguments>,
<fields>)
```

The `TRACEPOINT_EVENT` macro has four arguments: the tracepoint provider and event names (as above), the event arguments (a `TP_ARGS` macro), and the event fields (a `TP_FIELDS` macro). For any given `<provider name>, <event name>` combination, there can be only one `TRACEPOINT_EVENT`; in other words, you cannot overload event definitions.

4.1.2.3.1 `TP_ARGS`

`TP_ARGS` cannot be omitted but may be empty (`TP_ARGS()` or `TP_ARGS(void)`). It otherwise holds up to ten (`type, name`) pairs. The types may be primitive ones (`char, short, long, etc.`) or pointers to primitive ones (e.g. `char *, short *, long *, etc.`; this is also how arrays are passed in). Complex types (enums, structs, variants, etc.) are not yet possible.

4.1.2.3.2 `TP_FIELDS`

Like `TP_ARGS`, `TP_FIELDS` cannot be omitted but may be empty (`TP_FIELDS()` only; unlike `TP_ARGS`, you *cannot* use `TP_FIELDS(void)`). It otherwise holds a number of space-separated `ctf_*` calls that describe the event payload field names and their formatting according to Common Trace Format (CTF) conventions. They accept expressions for some of their arguments, so some payload manipulation can be handled by the tracepoint call itself (e.g. bit masking). Note in particular that while `TP_ARGS` defines what the `tracepoint()` call looks like for that event, `TP_FIELDS` defines what ends up in the trace event payload, and that the two are only loosely connected.

Known `ctf_*` calls are:

<code>ctf_integer</code>	Prints a (possibly signed) integer in decimal notation
<code>ctf_integer_nowrite</code>	
<code>ctf_integer_hex</code>	Prints an integer in hexadecimal notation
<code>ctf_integer_network</code>	
<code>ctf_integer_network_hex</code>	
<code>ctf_array</code>	Statically-sized array (of integers)
<code>ctf_array_nowrite</code>	
<code>ctf_array_text</code>	Prints text as an array of bytes
<code>ctf_array_text_nowrite</code>	
<code>ctf_sequence</code>	Dynamically-sized array (of integers)
<code>ctf_sequence_nowrite</code>	
<code>ctf_sequence_text</code>	Prints text as a sequence of bytes
<code>ctf_sequence_text_nowrite</code>	
<code>ctf_float</code>	Prints a floating-point number
<code>ctf_float_nowrite</code>	
<code>ctf_string</code>	Prints text as a character string
<code>ctf_string_nowrite</code>	

The difference between `_array` and `_sequence` is that the former is fixed (static) in size (number of elements) while the latter is dynamic in size. In practice this means an array's size is declared along with the array's type and is stored in the event metadata, while a sequence's size is looked up every time an instance is encountered and is stored in the payload. The `_hex` versions all store integers in the trace in the same way, but the metadata tells `babeltrace` to use hexadecimal notation instead of decimal (the default) when printing the value. The `_network` versions also all store the integers in the trace in the same way, but the metadata tells `babeltrace` to use network byte ordering when reading the value. Network byte ordering is most-significant byte (MSB) first and may or may not match the target's natural byte ordering, which is also specified in the metadata.

The `_nowrite` versions omit themselves from the session trace (including the metadata) but are otherwise identical. The `lttng list -f` command (see Section B.4.11) will mention them (labelled “[no write]”), but they won't be seen by `babeltrace` since they'll be absent from the event payload read from the trace. The `_nowrite` TP_FIELDS are meant to serve as a mechanism to make some of the event context available to the event filters without having to commit the data to the trace (see the `filter` option of the `enable-event` command, Section B.4.10). This is because an argument passed to `tracepoint()` is available for filter evaluation only if translated into an event payload field by a `ctf_*` invocation.

Note that the tracepoint facility does not provide any means of constructing enums, variants or structs. This is forthcoming (`ctf_enum*`, `ctf_struct*`, `ctf_variant*`), possibly with version 2.5 [24].

The `ctf_integer_*` calls all have the same signature: `(_type, _item, _src)`, where `_type` is an integer type⁶, `_item` is the field's identifier, and `_src` is a name from `TP_ARGS` or an expression that uses the names from `TP_ARGS`. The `_item` can be the same string as the `_src`.

The `ctf_float*` calls also have the `(_type, _item, _src)` signature, but this time `_type` is a floating-point type⁷. Currently, LTTng supports only the `float` and `double` floating-point types (complex numbers are wholly unsupported). Tracepoint provider code can use other floating-point types (such as `long double`), but the events using these will be rejected by the session daemon when the tracepoint provider registers its events (in earlier releases, `long double` fields would crash the tracer [25]). For example, in a typical tracing session, every time a process registers a tracepoint provider that includes unsupported events, you will get warning messages like this one (the parts in angle brackets will vary accordingly):

```
Warning: UST application '<process_name>' (pid: <PID>) has
unknown float mantissa '0' in field '<long_double_field_name>',
rejecting event '<provider_name>:<event_name>'
```

You will get this rejection message even if the process does not actually use the unsupported event, and you will also get it even when the session has expressly disabled it. The session daemon is rejecting the event prototype supplied by the tracepoint provider —this is completely separate from the session's per-event enable/disable settings.

Currently, the session daemon won't reject a `float _Complex` field, but the payload values will be interpreted as ordinary `floats`.

The unsupported floating-point types are not rejected at compilation time so that when LTTng does eventually start supporting these field types, the tracepoint providers won't need to be updated.

The `ctf_array*` calls all have the following signature: `(_type, _item, _src, _length)`. `_type`, `_item` and `_src` are as before while `_length` is the array's size (a constant or a constant expression). Note that arrays of arrays will have to be “unrolled” before being passed to the tracepoint call. The `ctf_array_text*` calls must use `char` for the `_type`.

⁶ The standard C integer types are `_Bool`, `char`, `signed char`, `unsigned short int`, `short int`, `unsigned long int`, `long int`, `unsigned long long int`, and `long long int`; `int` depends on the architecture, and your environment's shorthands are usually available (`short`, `long`, etc.).

⁷ The standard C floating-point types are `float`, `double`, and `long double`. The standard header `complex.h` adds the `float _Complex`, `double _Complex`, and `long double _Complex` floating-point types.

The `ctf_sequence*` calls all have the following signature: `(_type, _item, _src, _length_type, _src_length)`. `_type`, `_item` and `_src` are as before. `_length_type` must be an unsigned integer type (`unsigned short`, `unsigned long` or `unsigned long long`, even `size_t` may be used) otherwise the behaviour of the tracer is undefined —even if it does not crash, `babeltrace` will reject the resulting trace because it checks against a signed sequence length field (`babeltrace` will report ‘[error] Sequence length field should be `unsigned`.’). `_length_type`, finally, is like `_src` (i.e., a name from `TP_ARGS`). The `ctf_sequence_text*` calls must use `char` for the `_type`.

Finally, the `ctf_string*` calls both have this simple signature: `(_item, _src)`. `_item` and `_src` are as before. The `_src` should refer to a standard null-terminated C string.

The field name literals used in any of the `ctf_*` calls can be any standard C identifiers: arbitrary sequences of the characters a through z, A through Z, 0 through 9 (the digits), the underscore, and a long list of letter-like Unicode characters, except that the first character cannot be a digit (see Section 6.4.2 and Annex D of [23]). Using other literals (e.g. including a space or period within the field name) won’t cause any compilation nor tracing problems, but will result in a non-CTF-compliant trace which `babeltrace` won’t be able to read.

Example

Assume the following tracepoint event call:

```
tracepoint(ust_tests_hello, tptest, i, netint, values,
           text, strlen(text), dbl, flt);
```

Where `i`, `netint`, `values`, `text`, `dbl`, and `flt` are locally within scope. The tracepoint header may consist of:

```
TRACEPOINT_EVENT(ust_tests_hello, tptest,
    TP_ARGS (int,      anint,      int,      netint, long *, values,
              char *,     text,      size_t,   textlen,
              double,    doublearg, float,   floatarg),
    TP_FIELDS(
        ctf_integer(long, longfield, anint)
        ctf_integer_hex(int, intfield2, anint)
        ctf_integer_network_hex(int, netintfieldhex, netint)
        ctf_array(long, arrfield1, values, 3)
        ctf_sequence_text(char, seqfield2, text, size_t, textlen)
        ctf_float(float, floatfield, floatarg)
        ctf_float(double, doublefield, doublearg)
    )
)
TRACEPOINT_LOGLEVEL(ust_tests_hello, tptest, TRACE_DEBUG_UNIT)
TRACEPOINT_MODEL_EMF_URI(ust_tests_hello, tptest, \
    "http://example.com/path_to_model?q=ust_tests_hello:tptest")
```

The event payload would consist of an int in two fields (`longfield` padding it to long width and `intfield2` an int-wide hex string), another int-wide hex string (`netintfieldhex`, whose bytes are in network transmission order), an array of three long values (`arrfield1`), a string (`seqfield2`) of `textlen` characters, a float (`floatfield`) and a double (`doublefield`). Note that the `TP_FIELDS` declarations are not comma-separated.

4.1.2.4 The `TRACEPOINT_LOGLEVEL` macro

```
TRACEPOINT_LOGLEVEL(<provider name>, <event name>, <loglevel>)
```

The `TRACEPOINT_LOGLEVEL` macro has three arguments: the tracepoint provider name, the event name, and the logging level name (`TRACE_*` constants listed in `tracepoint.h`):

<i>Loglevel</i>	<i>Shorthand</i>	<i>Description</i>
0 <code>TRACE_EMERG</code>	<code>EMERG</code>	system is unusable
1 <code>TRACE_ALERT</code>	<code>ALERT</code>	action must be taken immediately
2 <code>TRACE_CRIT</code>	<code>CRIT</code>	critical conditions
3 <code>TRACE_ERR</code>	<code>ERR</code>	error conditions
4 <code>TRACE_WARNING</code>	<code>WARNING</code>	warning conditions
5 <code>TRACE_NOTICE</code>	<code>NOTICE</code>	normal, but significant, condition
6 <code>TRACE_INFO</code>	<code>INFO</code>	informational message
7 <code>TRACE_DEBUG_SYSTEM</code>	<code>SYSTEM</code>	system-level scope (set of programs)
8 <code>TRACE_DEBUG_PROGRAM</code>	<code>PROGRAM</code>	program-level scope (set of processes)
9 <code>TRACE_DEBUG_PROCESS</code>	<code>PROCESS</code>	process-level scope (set of modules)
10 <code>TRACE_DEBUG_MODULE</code>	<code>MODULE</code>	module (executable/library) scope (set of units)
11 <code>TRACE_DEBUG_UNIT</code>	<code>UNIT</code>	compilation unit scope (set of functions)
12 <code>TRACE_DEBUG_FUNCTION</code>	<code>FUNCTION</code>	function-level scope
13 <code>TRACE_DEBUG_LINE</code>	<code>LINE</code>	line-level scope (<code>TRACEPOINT_EVENT</code> default)
14 <code>TRACE_DEBUG</code>	<code>DEBUG</code>	debug-level message (<code>trace_printf</code> default)

Higher loglevel numbers imply the most verbosity (expect higher tracing throughput). Loglevels 0 through 6 and loglevel 14 match `syslog(3)` level semantics. Loglevels 7 through 13 offer more fine-grained selection of debug information.

4.1.2.5 The `TRACEPOINT_MODEL_EMF_URI` macro

```
TRACEPOINT_MODEL_EMF_URI(<provider name>, <event name>, "<URI>")
```

The `TRACEPOINT_MODEL_EMF_URI` macro has three arguments: the tracepoint provider name, the event name, and the URI (Uniform Resource Identifier) of the tracepoint event's EMF (Eclipse Modeling Framework) model.

4.1.2.6 The TRACEPOINT_EVENT_CLASS and TRACEPOINT_EVENT_INSTANCE macros

Whereas each TRACEPOINT_EVENT macro defines a single tracepoint, a TRACEPOINT_EVENT_CLASS macro defines an abstract tracepoint, a *template* which is used by any number of later TRACEPOINT_EVENT_INSTANCE macros to define actual tracepoints. Using TRACEPOINT_EVENT_CLASS and TRACEPOINT_EVENT_INSTANCE makes it possible to succinctly define families of tracepoints that are identical except for their names.

The TRACEPOINT_EVENT_CLASS macros are written in precisely the same way as the TRACEPOINT_EVENT macros. The syntax of TRACEPOINT_EVENT_INSTANCE is:

```
TRACEPOINT_EVENT_INSTANCE(<provider name>, <template name>,
<event name>, <arguments>)
```

If the set of TP_ARGS supplied to a TRACEPOINT_EVENT_INSTANCE declaration does not match the template's set (this includes even a simple re-ordering of the arguments), there is no compilation warning but the resulting tracepoint provider will issue one upon being loaded: "Warning: Tracepoint signature mismatch, not enabling one or more tracepoints [...]" . The incorrectly declared TRACEPOINT_EVENT_INSTANCE events won't be enabled.

4.1.3 Building instrumented software applications

When building the instrumented application, there are two choices. The tracepoint provider can be included in the application (statically linked), or it can be dynamically linked. The example given at the beginning of Section 4.1 is statically linked.

4.1.3.1 Static linking

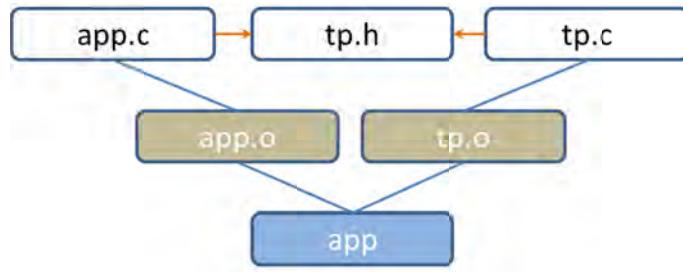


Figure 31: Building a statically linked instrumented application.
Source code in white; include directives shown as orange arrows; object files in tawny; executable in blue. tp.c and tp.h define the tracepoint provider (tp.o).

Static linking is more portable, but does increase the client application's size slightly as well as force a recompilation whenever the tracepoint provider is modified. The tracepoint provider can be included directly in the client (Figure 31), or a static library (.a) can be used (Figure 32). This makes no difference to the client (the executables will be binary-identical), because a static library (.a) is nothing else than an archive of object files (.o). When a client is linked against a static library, the machine code from the object files for the tracepoint provider functions used by the client is *copied* from the library into the final executable —which is precisely what also happens when the tracepoint provider is included directly (treated as just one more module within the client's source tree).

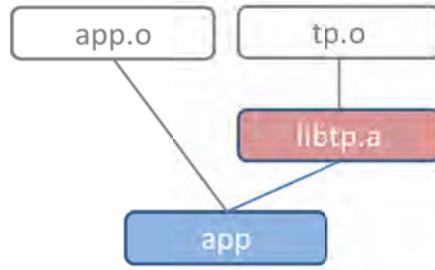


Figure 32: Building an instrumented application using a static library.
Static library (archive) in salmon.

Using a static library does have the advantage of centralising the tracepoint provider object so it can be shared between multiple clients. This way, when the tracepoint provider is modified, the source code changes don't have to be patched into each client's source code tree. The clients need to be relinked after each change, but need not be otherwise recompiled (unless the tracepoint provider's API changes, of course).

Recall the makefile from the beginning of Section 4.1, which produces the statically linked instrumented application `sample_static`:

```
CC = gcc
LIBDL = -ldl # On Linux
LIBUST = -littng-ust
#LIBS = $(LIBDL) $(LIBUST)
LOCAL_CPPFLAGS += -I.
LDFLAGS += -L/usr/local/lib
TP_DEFINE = -D TRACEPOINT_DEFINE

all: static_samples

static_samples: static

#####
#Trace provider statically included.
static: static.o tp.o
    @echo "~~~~~Linking sample_@"
    $(CC) -o sample_@ $^ $(LDFLAGS) $(LIBDL) $(LIBUST)
    @echo "    Use './sample_@' to run sample_@"

static.o: sample.c tp.h
    @echo "~~~~~Compiling @@
    $(CC) $(CPPFLAGS) $(LOCAL_CPPFLAGS) $(TP_DEFINE) -c -o $@ $<

tp.o: tp.c tp.h
    @echo "~~~~~Compiling @@
    $(CC) $(CPPFLAGS) $(LOCAL_CPPFLAGS) -c -o $@ $<
```

A `static_lib` target, producing `sample_static_lib`, an instrumented application prepared using a static library, can be added with these few lines (additions in red):

```
AR = ar
CC = gcc
LIBDL = -ldl # On Linux
LIBUST = -lltng-ust
#LIBS = $(LIBDL) $(LIBUST)
LOCAL_CPPFLAGS += -I.
LDFLAGS += -L/usr/local/lib
TP_DEFINE = -D TRACEPOINT_DEFINE

all: static_samples

static_samples: static static_lib

#####
#Trace provider statically included.
static: static.o tp.o
    @echo "~~~~~Linking sample_@"
    $(CC) -o sample_$@ $^ $(LDFLAGS) $(LIBDL) $(LIBUST)
    @echo "    Use './sample_$@' to run sample_@"

#####
#Trace provider statically included through a static library.
#The resulting executable is identical to that of the 'static' target.
static_lib: static.o tp.a
    @echo "~~~~~Linking sample_@"
    $(CC) -o sample_$@ $^ $(LDFLAGS) $(LIBDL) $(LIBUST)
    @echo "    Use './sample_$@' to run sample_@"

static.o: sample.c tp.h
    @echo "~~~~~Compiling @@
    $(CC) $(CPPFLAGS) $(LOCAL_CPPFLAGS) $(TP_DEFINE) -c -o $@ $<

tp.a: tp.o
    @echo "~~~~~Packaging @@
    $(AR) -cq $@ $<

tp.o: tp.c tp.h
    @echo "~~~~~Compiling @@
    $(CC) $(CPPFLAGS) $(LOCAL_CPPFLAGS) -c -o $@ $<
```

Note how the intermediate targets `static.o` and `tp.o` are shared between the two ultimate targets `static` and `static_lib`. As can be seen, the static library `tp.a` can be freely substituted for the compiled object `tp.o`, and is prepared using the standard archiving application `ar`.

4.1.3.2 Statically-aware dynamic linking

Dynamic linking makes the client application binaries largely independent of the tracepoint provider. As in the static case, this is true as long as the provider's API does not change (i.e. changing the number of arguments in the `tracepoint()` call or their types breaks this contract and forces a rewrite and subsequent recompilation of the clients). The tracepoint binary can be recompiled (after changes are made to the event payload fields, for example) into a new dynamic library (`.so`), and the clients will use the modified binary the next time they are run.

There is a slight initial performance cost when running a dynamically linked client as opposed to a statically linked one, since the system is loading the dynamic library on demand. There is an improvement in client size, since each client binary no longer includes the tracepoint provider object image. A more serious disadvantage is that the clients become unusable (to a greater or lesser degree) if the dynamic library isn't available (or cannot be found) on the target system. This is easily eclipsed by the great flexibility that dynamic linking offers: applications can, sometimes even while running, swap dynamic libraries in and out, diminishing an application's memory footprint. Maintenance of the application is also greatly enhanced, as new versions of the dynamic libraries can be installed without needing to re-install the rest. Plug-in architectures are usually realised using dynamic libraries.

Dynamic linking, like static linking, comes in two flavours. The first is static awareness (Figure 33): the dynamic library objects are not included into the executable but are tied to the execution. You can see which libraries an ELF (Executable and Linkable Format) executable depends on by reading its `NEEDED` tags, using `readelf -d application | grep NEEDED`. The executable expects the `NEEDED` libraries to be present in its environment; the operating system obliges by having the dynamic loader (see Topical Note A.13) look in a predefined set of folders, the *library search path*, for the requested dynamic libraries (this is similar to, but distinct from, the *executable search path* used by the shell to find the executable that a command invokes). If one of the requested dynamic libraries is not found, an error is issued (`cannot open shared object file`) and the executable does not run at all.

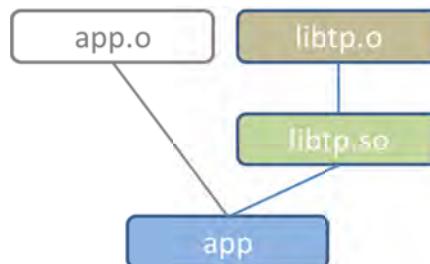


Figure 33: Building a dynamically linked, statically aware instrumented application.
Dynamic library in olive. `libtp.o` is just `tp.o` with an extra compilation flag.

LTTng normally does not instrument applications in this way: the approach shown in the next section is more usual. Here the `static_aware` target is added to the makefile in order to make a `sample_static_aware` instrumented application:

```
AR = ar
CC = gcc
LIBDL = -ldl # On Linux
LIBUST = -llttng-ust
#LIBS = $(LIBDL) $(LIBUST)
LOCAL_CPPFLAGS += -I.
LDFLAGS += -L/usr/local/lib
SOFLAGS = -fPIC
TP_DEFINE = -D TRACEPOINT_DEFINE
SOVERSION_MAJOR = 1
SOVERSION_MINOR = 0

all: static_samples

static_samples: static static_lib static_aware
[...]

tp.o: tp.c tp.h
    @echo "~~~~~Compiling $@:"
    $(CC) $(CPPFLAGS) $(LOCAL_CPPFLAGS) -c -o $@ $<
#####
#Trace provider as a dynamic library.
libtp.so: libtp.o
    @echo "~~~~~Packaging $@:"
    $(CC) -shared -Wl,-soname,$@.$(SOVERSION_MAJOR) \
        -Wl,-no-as-needed -o \
        $@.$(SOVERSION_MAJOR).$(SOVERSION_MINOR) $(LDFLAGS) \
        $(LIBUST) $<
    ln -sf $@.$(SOVERSION_MAJOR).$(SOVERSION_MINOR) \
        $@.$(SOVERSION_MAJOR)
    ln -sf $@.$(SOVERSION_MAJOR) $@

libtp.o: tp.c tp.h
    @echo "~~~~~Compiling $@:"
    $(CC) $(CPPFLAGS) $(LOCAL_CPPFLAGS) $(SOFLAGS) -c -o $@ $<
```

```
#####
#Trace provider (libtp) dynamically included as a dependency.
#Note that "-L. -ltp", when used, should appear before
#$(LDFLAGS) $(LIBDL).
static_aware: static.o libtp.so
    @echo "~~~~~Linking sample_$@:"

    $(CC) -Wl,-no-as-needed -o sample_$@ -L. -ltp $< \
        $(LDFLAGS) $(LIBDL)
    @echo "  Use 'LD_PRELOAD=./libtp.so ./sample_$@' to " \
        "run sample_$@"
```

The first addition is the `libtp.so` intermediate target, which produces the dynamic library (shared object) of the same name. This essentially replaces `tp.o` in the dynamic linking case and will be re-used extensively later on. The key difference (highlighted in yellow) between `tp.o` and `libtp.o` is the `fPIC` compiler flag, which forces the `libtp.o` output to be in “position-independent code” (PIC), a requirement of shared libraries. The shared library itself (`libtp.so`) is created using the `shared` compiler flag (also highlighted in yellow).

The `soname` linker option (passed to the linker by the `Wl` option) serves to set the library’s name to `libtp.so.1`, where `1` is its major version number (highlighted in green). The two `ln` instructions are optional and merely serve to create symbolic links (`ln -sf`) to the same shared library so it can be accessed using its unqualified name (`libtp.so`), its major-versioned name (`libtp.so.1`) or its minor-versioned name (`libtp.so.1.0`). This is standard Linux practice.

The `sample_static_aware` application is linked to the shared library with the addition of the `ltp` option (highlighted in grey); the option is `1`, its argument `tp`, implicitly prefixed by `lib` and suffixed by `.so`, meaning “link the application to `libtp.so`”. This is what makes it statically aware of the library.

As stated earlier, static awareness is realised by the addition of `NEEDED` tags to the executable. If an object is compiled with the `as-needed` flag, the libraries mentioned in the command line (`LIBDL` and `libtp` in one case and `LIBUST` in the other) generate a `NEEDED` tag in the executable *only if actually satisfying an otherwise undefined symbol reference*. It turns out LTTng implements its tracepoints in such a way that it does not generate symbol references to `libtp` or `LIBUST`. The `no-as-needed` flag (highlighted in blue) is used to force the `NEEDED` tags to appear. Since `as-needed` could be set by the environment as a default, compiling without this flag would result in a `libtp` that cannot call on `LIBUST` and an application that cannot call on `libtp`. The application would remain runnable, but tracing would become impossible.

If the tracepoint provider code were modified to add some methods to it, and these methods were explicitly called from the application, the `no-as-needed` flag would become unnecessary. The same effect would be achieved if these extra methods, compiled as a separate object, were bundled into the tracepoint provider’s dynamic library.

Now that the `sample_static_aware` application has been made statically aware of `libtp` and cannot run without it, how does one get it to run? Several techniques are possible. The `LD_PRELOAD` technique recommended by the makefile has `libtp` preloaded by the command line before invoking the application. The only care needed here is in making sure the path to `libtp` is correct. A second technique consists in using the much-maligned `LD_LIBRARY_PATH` environment variable. Finally, the static-aware compiler call could be replaced by this one:

```
$ (CC) -Wl,-no-as-needed -o sample_$@ -L. -ltp $< \
$ (LDFLAGS) $(LIBDL) -Wl,-rpath,'$$ORIGIN',--enable-new-dtags
```

This adds an `RPATH` tag to the application which makes it look for its dependencies in its starting folder ('`$$ORIGIN`'). The `sample_static_aware` application could then be run without any additional commands in its command line. `RPATH` is, unlike `RUNPATH`, applicable to libraries as well: a library with an `RPATH` tag will look there for its dependencies; failing its own `RPATH`, it will use its executable's `RPATH`.

The `LD_PRELOAD` technique is the only one that will work in the `dynamic` target case discussed next.

4.1.3.3 Fully dynamic linking

The second dynamic linking approach is full dynamicity (Figure 34): the library and its member functions are explicitly sought, loaded and unloaded at runtime, using the `libdl` library (`dlopen`, `dlsym`, `dlclose`). The shared library is dynamically *loaded*, as opposed to dynamically *linked* as in the previous case. For a variety of reasons, LTTng's tracepoint providers are dynamically loaded (which is why they do not generate symbol references to `libtp` or `LIBUST`). The tracepoint provider library loading is handled by the `lttng-ust` library: if the library is not present when the application is started, the tracepoint calls become inert. Otherwise, they are available for tracing. The application can control the availability of its tracepoints by using `dlopen` and `dlclose` itself.

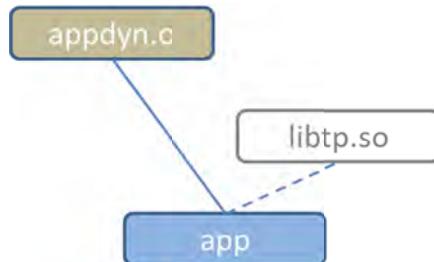


Figure 34: Building an instrumented application using dynamic loading.

`appdyn.o` is just `app.o` with an added source code switch (which affects only `lttng/tracepoint.h`, invoked from `tp.h`). Compilation of the `app` need not be aware of `libtp.so`: it is sought at runtime. One could also write a distinct `appdyn.c` by adding explicit `dlopen` calls to the original `app.c`.

Here the `dynamic` target is added to the makefile in order to make a `sample_dynamic` instrumented application:

```
AR = ar
CC = gcc
LIBDL = -ldl # On Linux
LIBUST = -lltng-ust
#LIBS = $(LIBDL) $(LIBUST)
LOCAL_CPPFLAGS += -I.
LDFLAGS += -L/usr/local/lib
SOFLAGS = -fPIC
TP_DEFINE = -D TRACEPOINT_DEFINE
TP_DEFINE_DYNAMIC = $(TP_DEFINE) -D TRACEPOINT_PROBE_DYNAMIC_LINKAGE
SOVERSION_MAJOR = 1
SOVERSION_MINOR = 0

all: static_samples dynamic_samples

static_samples: static static_lib static_aware

dynamic_samples: dynamic
[...]

static.o: sample.c tp.h
@echo "~~~~~Compiling $@:"
$(CC) $(CPPFLAGS) $(LOCAL_CPPFLAGS) $(TP_DEFINE) -c -o $@ $<
[...]

libtp.o: tp.c tp.h
@echo "~~~~~Compiling $@:"
$(CC) $(CPPFLAGS) $(LOCAL_CPPFLAGS) $(SOFLAGS) -c -o $@ $<

#####
#Trace provider (libtp) dynamically included as a dependency.
#Note that "-L. -ltp", when used, should appear before
#$(LDFLAGS) $(LIBDL).
static_aware: static.o libtp.so
@echo "~~~~~Linking sample_$@:"
$(CC) -Wl,-no-as-needed -o sample_$@ -L. -ltp $< \
$(LDFLAGS) $(LIBDL)
@echo "    Use 'LD_PRELOAD=./libtp.so ./sample_$@' to " \
"run sample_$@"
```

```

#####
#Trace provider (libtp) dynamically included (no static awareness).
#If not preloaded, the sample application runs but won't be traceable.
#This is indifferent to the -[no-]as-needed linker flag.
dynamic: dynamic.o libtp.so
        @echo "~~~~~Linking sample_#{@:"
        $(CC) -o sample_#{@ $< $(LDFLAGS) $(LIBDL)
        @echo "  Use '[LD_PRELOAD=./libtp.so] ./sample_#{@' to "
        "run sample_#{@"

#Compare with the 'static.o' target.
dynamic.o: sample.c tp.h
        @echo "~~~~~Compiling #{@:"
        $(CC) $(CPPFLAGS) $(LOCAL_CPPFLAGS) $(TP_DEFINE_DYNAMIC) \
        -c -o #{@ $<

```

The application's main module is compiled with one little change: the `$(TP_DEFINE)` flag is replaced with a `$(TP_DEFINE_DYNAMIC)` flag (highlighted in yellow), which merely adds the `TRACEPOINT_PROBE_DYNAMIC_LINKAGE` define, discussed earlier. The `dynamic` target, compared to the `static` target, drops the `no-as-needed` and `ltp` flags, which are no longer needed since the application need not to be statically aware of the `libtp` dynamic library.

The application runs regardless of the presence of the tracepoint provider's dynamic library. To have tracing happen, however, you must preload it. This is because the tracepoint becomes inert if LTTng fails to find `libtp` in the application's memory space.

The instrumented application can take an active role in controlling its traceability. First, `sample.c` is modified so that it compiles differently when the `SAMPLE_DLOPEN` compiler flag is set:

```
#include <stdio.h> //For printf
#include <unistd.h>
#include "tp.h"

int main(int argc, char **argv)
{
    int i = 0;
    char themessage[20]; //Can hold up to "Hello World 9999999\0"
#ifndef SAMPLE_DLOPEN
    void *libtp_handle;

    //You must use one of RTLD_LAZY or RTLD_NOW
    //RTLD_NOLOAD prevents incrementation of the use count
    libtp_handle = dlopen("./libtp.so", RTLD_LAZY | RTLD_NOLOAD);
    if (libtp_handle) {
        fprintf(stderr, "libtp resident (preloaded)\n");
    } else {
        fprintf(stderr, "libtp not resident\n");
        libtp_handle = dlopen("./libtp.so", RTLD_LAZY);
        if (!libtp_handle) {
            fprintf(stderr, "libtp not found nor loaded\n");
        } else {
            fprintf(stderr, "libtp found and loaded\n");
        }
    }
#endif

    fprintf(stderr, "sample starting\n");
    for (i = 0; i < 10000; i++) {
#ifndef SAMPLE_DLOPEN
        if ((i == 3333) && (libtp_handle)) \
            dlclose(libtp_handle);
        if (i == 6666) \
            libtp_handle = dlopen("./libtp.so", RTLD_LAZY);
#endif
        sprintf(themessage, "Hello World %u", i);
        tracepoint(sample_component, event, themessage);
        usleep(1);
    }
    fprintf(stderr, "sample done\n");
#endif
    if (libtp_handle) return dlclose(libtp_handle);
#endif
    return 0;
}
```

The block before the loop retrieves `libtp` from memory (if preloaded) or disc (otherwise). During the loop, a third of the way through, the application tries to unload `libtp` (this will fail if it was preloaded, even though `dlclose` still “succeeds”). Two-thirds of the way through, the application reloads `libtp`. If the dynamic library was not preloaded and is found at the expected path, the effect on tracing will be that the application will register itself as an event source with LTTng, then issue a bunch of events, then unregister itself, only to register itself anew a little while later, issuing another bunch of events before shutting down. A typical per-process-ID trace will see this as two separate occurrences of event-issuing processes that just happen to have the same process ID. Process IDs are eventually re-used by Linux under normal conditions once there is a sufficient rollover of processes, but this usually takes quite a while.

The `dynamic_dlopen` target is added to the makefile:

```
AR = ar
CC = gcc
LIBDL = -ldl # On Linux
LIBUST = -littng-ust
#LIBS = $(LIBDL) $(LIBUST)
LOCAL_CPPFLAGS += -I.
LDFLAGS += -L/usr/local/lib
SOFLAGS = -fPIC
TP_DEFINE = -D TRACEPOINT_DEFINE
TP_DEFINE_DYNAMIC = $(TP_DEFINE) -D TRACEPOINT_PROBE_DYNAMIC_LINKAGE
USE_DOPEN = -D SAMPLE_DOPEN
SOVERSION_MAJOR = 1
SOVERSION_MINOR = 0

all: static_samples dynamic_samples

static_samples: static static_lib static_aware

dynamic_samples: dynamic dynamic_dlopen

[...]

#####
#Trace provider (libtp) dynamically included (no static awareness).
#If not preloaded, the sample application runs but won't be traceable.
#This is indifferent to the -[no-]as-needed linker flag.
dynamic: dynamic.o libtp.so
    @echo "~~~~~Linking sample_$@:"
    $(CC) -o sample_$@ $< $(LDFLAGS) $(LIBDL)
    @echo "    Use '[LD_PRELOAD=./libtp.so] ./sample_$@' to "
    "run sample_$@"

dynamic.o: sample.c tp.h
    @echo "~~~~~Compiling $@:"
    $(CC) $(CPPFLAGS) $(LOCAL_CPPFLAGS) $(TP_DEFINE_DYNAMIC) \
        -c -o $@ $<
```

```

#####
#Trace provider (libtp) dynamically included (no static awareness).
#The sample application attempts to unload libtp during its run, only
#to later reload it.
#This is indifferent to the -[no-]as-needed linker flag.
dynamic_dlopen: dynamic_dlopen.o libtp.so
    @echo "~~~~~Linking sample_@"
    $(CC) -o sample_$@ $< $(LDFLAGS) $(LIBDL)
    @echo "  Use '[LD_PRELOAD=./libtp.so] ./sample_$@' to "
    "run sample_@"

dynamic_dlopen.o: sample.c tp.h
    @echo "~~~~~Compiling $@:"
    $(CC) $(CPPFLAGS) $(LOCAL_CPPFLAGS) $(TP_DEFINE_DYNAMIC) \
        $(USE_DLOPEN) -c -o $@ $<

```

As can be seen, the only real difference is the use of `$(USE_DLOPEN)` to change the application's main module (highlighted in yellow).

The `/usr/src/lttng-ust/doc/examples/drdc` folder's Makefile and assorted source files contains much more than the fragments discussed so far. It is recommended to take the time to browse through them and run the various `sample` executables. For instance, the `dynamic_lib` target illustrates how the `$(TP_DEFINE)` compiler switch must be used once per “module”: it builds an application from four compilation units, all instrumented: two units are bundled together in a dynamic library while the other two are bundled in the executable. The tracepoint header (`tp.h`) is included in all four units while the `$(TP_DEFINE)` compiler switch is turned on only twice: once for the executable's main compilation unit, and once in one of the two compilation units destined for the dynamic library.

4.2 Instrumenting a system function for just one application

`lttng-ust/liblttng-ust-libc-wrapper/README` explains how to very simply wrap the target system call to instrument it. A small `.so` is created which is then preloaded (`LD_PRELOAD`) when calling the application, so that the application sees the specially-wrought system call instead of the “true” one.

4.3 Instrumenting a C++ application (user-space)

To trace a C++ application, see the `lttng-ust/tests/hello.cxx` folder⁸. It contains a small C++ application, `hello`, and its tracepoint provider `ust_tests_hello`. The instrumentation is done in precisely the same way as for a C application.

⁸ The conventional C++ file extension is ‘cxx’, because ‘cpp’ is assigned to the “C pre-processor”.

C++ can readily be instrumented, because of its cross-compatibility with C. In this section, the examples used throughout Section 4.1 will be used as a starting point. The `sample.c` source is first modified to call on a dynamic library written in C++. It is possible to do this from a pure C application, although one must occasionally wrestle with “name mangling” to achieve this. To facilitate the process, `sample.c` is turned into `sample.cxx`:

```

#include <stdio.h> //For printf
#include <unistd.h>
#include "tp.h"
#include "cxxobject.hxx"

int main(int argc, char **argv)
{
    int i = 0;
    char themessage[20]; //Can hold up to "Hello World 9999999\0"
#ifndef SAMPLE_DLOPEN
    void *libtp_handle;

    //You must use one of RTLD_LAZY or RTLD_NOW
    //RTLD_NOLOAD prevents incrementation of the use count
    //Opening "libtp.so" means we rely on the search path, which
    //is why sample.cxx is compiled with an RPATH tag.
    libtp_handle = dlopen("libtp.so", RTLD_LAZY | RTLD_NOLOAD);
    if (libtp_handle) {
        fprintf(stderr, "libtp resident (preloaded)\n");
    } else {
        fprintf(stderr, "libtp not resident\n");
        libtp_handle = dlopen("libtp.so", RTLD_LAZY);
        if (!libtp_handle) {
            fprintf(stderr, "libtp not found nor loaded\n");
        } else {
            fprintf(stderr, "libtp found and loaded\n");
        }
    }
#endif
/*
 * This source code is written in C except for this line, the C++
 * object's constructor invocation. This and the inclusion of
 * cxx_sample.hxx makes this source C++, hence "sample.cxx".
 */
CXXObject cxxo(42);

fprintf(stderr, "sample.cxx starting\n");
for (i = 0; i < 10000; i++) {
#ifndef SAMPLE_DLOPEN
    if ((i == 3333) && (libtp_handle)) dlclose(libtp_handle);
    if (i == 6666) \
        libtp_handle = dlopen("libtp.so", RTLD_LAZY);
#endif
    cxxo.setvalue(i);
    cxxo.message(themessage);
    tracepoint(sample_component, event, themessage);
    usleep(1);
}
fprintf(stderr, "sample.cxx done\n");

```

```

#define SAMPLE_DLOPEN
    if (libtp_handle) return dlclose(libtp_handle);
#endif
    return 0;
}

```

Besides using a C++ object to prepare the message sent through the tracepoint, this code expects to be compiled with an `RPATH` tag as explained at the end of Section 4.1.3.2. If the C++ object, instead of having a `message` method that manipulates a C string, were to expose a C++ `std::string` (called `somestring` in the code fragment below), this would need to be converted to a C string like so:

```

tracepoint(sample_component, event, \
           (char *) (cxxo.somestring().c_str()));

```

Here is the C++ header (`cxxobject.hxx`):

```

#include <iostream>
#include <string>

class CXXObject
{
public:
    CXXObject(int value);
    void message(char *themessage);
    int getvalue();
    void setvalue(int value);
private:
    int somevalue;
};

```

Here is the C++ source (`cxxobject.cpp`):

```
#include "cxxobject.hxx"
#include <cstring> //for strcpy
#include <iostream>
#include <sstream> //for ostringstream
#include <string>

#include "tp.h"

CXXObject::CXXObject(int value)
{
    //constructor
    somevalue = value;
}

void CXXObject::message(char *themessage)
{
    std::ostringstream oss;
    oss << "CXXObject " << somevalue;
    strcpy(themessage, oss.str().c_str());
    tracepoint(sample_component, event,
               \const_cast<char *>("CXXObject::message"));
}

int CXXObject::getvalue()
{
    return somevalue;
}

void CXXObject::setvalue(int value)
{
    somevalue = value;
}
```

The message method can be rewritten to return a `std::string` like so (`cxxobject.hxx` would need to be modified accordingly, of course):

```
std::string CXXObject::message()
{
    std::ostringstream oss;
    oss << "CXXObject " << somevalue;
    tracepoint(sample_component, event,
               \const_cast<char *>("CXXObject::message"));
    return oss.str();
}
```

Adding a `dynamic_cxx` target to the Makefile:

```
AR = ar
CC = gcc
CXX = g++
[...]

#####
#Trace provider (libtp) dynamically included (no static awareness).
#This sample application uses a C++ instrumented dynamic library.
#Setting -Wl,-rpath,'$$ORIGIN' allows the sample to readily find
#the non-optional libcxxobject.
dynamic_cxx: dynamic_cxx.o libcxxobject.so
    @echo "~~~~~Linking sample_@"
    $(CXX) -Wl,-no-as-needed -o sample_@ -L. -lcxxobject $< \
        $(LDFLAGS) $(LIBDL) -Wl,-rpath,'$$ORIGIN',--enable-new-dtags
    @echo "    Use '[LD_PRELOAD=./libtp.so] ./sample_@' to " \
        "run sample_@"

dynamic_cxx.o: sample.cxx cxxobject.hxx tp.h
    @echo "~~~~~Compiling @@
    $(CXX) $(TP_DEFINE_DYNAMIC) \
        -c -o $@ $<

#Note that $(CC) does just fine, no need for $(CXX)
#This is indifferent to the -[no-]as-needed linker flag.
libcxxobject.so: libcxxobject.o
    @echo "~~~~~Packaging @@
    $(CC) -shared -Wl,-soname,$@.$(SOVERSION_MAJOR) \
        -Wl,-no-as-needed -o \
        $@.$(SOVERSION_MAJOR).$(SOVERSION_MINOR) $<
    ln -sf $@.$(SOVERSION_MAJOR).$(SOVERSION_MINOR) \
        $@.$(SOVERSION_MAJOR)
    ln -sf $@.$(SOVERSION_MAJOR) $@

libcxxobject.o: cxxobject.cxx cxxobject.hxx tp.h
    @echo "~~~~~Compiling @@
    $(CXX) $(SOFLAGS) \
        $(TP_DEFINE_DYNAMIC) -c -o $@ $<
```

Observe that the C++ compiler is not required to repackage object files as a shared object. This is because the object file format is independent of the language the code was compiled from.

4.4 Instrumenting a Java application

Java applications are virtual in the sense that they execute on a virtual machine (the Java Virtual Machine, JVM) which has limited access to its host. While LTTng kernel and user-space probes are embedded in object code (streams of CPU instructions), a Java application's byte code is understandable only by the JVM. Instrumenting a Java application can thus be done at three levels. First, at the Java source code level: the application can be explicitly instrumented, typically using the `java.util.logging` API. The instrumentation groundwork need not be LTTng-specific, as the SLF4J example will show later on. Second, at the byte code level: the `java.lang.instrumentation` API can be used to plug a tracing tool into an already-running JVM and to instrument/de-instrument various classes by the injection of byte code (e.g. to trace entry into and exit from methods). Finally, at the JVM's executable code level: the JVM itself could be instrumented.

This last option is ambitious as it entails rewriting the source code of the JVM in order to insert tracepoints, very much like what was done with the Linux kernel. The problem is that several different JVMs exist, some of which have licenses that preclude their modification in this way or don't make the source code available. Not all JVMs are written in C or C++ either. The tracepoints would need to be maintained between releases of the JVM, a tall order. Such instrumentation would mostly be of interest for people wishing to study how the JVM behaves, and would have little relevance to the study of a specific Java application. Arguably, a lot (if not all) of that instrumentation can be achieved outside of the JVM, by tracing the system and native calls it makes.

Going back to the first level of instrumentation, a Java application can be instrumented by adding Java tracepoint statements which are methods of a light-weight Java class. This is similar to the way C or C++ applications have tracepoints added to them. The light-weight Java class triggers the corresponding C or C++ trace provider calls (using JNI, the Java Native Interface [26] [27], or JNA, Java Native Access [28]). The execution stream is, in effect, stepping out of the JVM and into an external library just long enough to trip a tracepoint in the library.

Tracepoints can be integrated into the existing Java logging facilities using an adaptor layer such as SLF4J (Simple Logging Façade for Java; see <http://www.slf4j.org/>). For example:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HelloWorld {
    public static void main(String[] args) {
        Logger logger = LoggerFactory.getLogger(HelloWorld.class);
        logger.info("Hello World");
    }
}
```

Applications instrumented using SLF4J require only that `slf4j-api-1.7.5.jar` be in the class path; at run-time, SLF4J looks for a binding of its `org.slf4j.Logger` interface and uses that to implement logging (or else SLF4J supplies a no-operation default binding). This allows swapping loggers in and out simply by changing the classpath or the classpath's contents.

4.4.1 JNI and tracepoint providers

JNI's main focus is to export the implementation of methods invoked on Java objects to an external library written in some other language. In this perspective, JNI assumes Java comes first and the external library is written explicitly in order to support the Java application. In that sense, JNI is not designed to provide a simple way to write Java wrappers for pre-existing external libraries. Thin adaptor libraries, invoked through JNI, are possible as long as the legacy library's interface involves reasonably primitive argument types. Typically, some string manipulation and memory management is needed.

An adaptor library is not a practical solution with LTTng. Tracepoint provider code (see `tp.c` and `tp.h` in Sections 4.1.2.1 and 4.1.2.2) makes heavy use of macros and contextual compilation flags, and the resulting libraries have no readily recognisable entry point for the tracepoints themselves. It is much easier to recycle the tracepoint header into a new tracepoint provider library with built-in JNI support, as will be seen shortly. In fact, this can be made a standard operational procedure, as the addition of JNI support leaves the library still perfectly usable from more conventional contexts such as C or C++ projects. The only drawback to this approach is that a new Java wrapper class will need to be created for each tracepoint provider library. This is not much of a problem since most of the Java wrapper code will be boilerplate.

Whereas JNI allows one to write supporting libraries for Java, JNA (Java Native Access, available on Ubuntu 12.04.3 LTS as the `libjna-java` package) focuses on giving Java direct access to existing external libraries. JNA uses a small native library called the foreign function interface library (`libffi`) to dynamically invoke native code. Using JNA, one creates a Java interface that describes the functions and structures of the target native library, without static bindings, header files, or any compile phase. JNA could be a much better match than JNI for the relatively simple task of invoking tracepoints from Java.

Could a generic Java class be written that would allow dynamic access to *any* tracepoint provider library? JNI supplies a rich set of accessor functions that allow the external library to manipulate the Java objects and indeed the entire JVM instance (an external library can handle or throw exceptions, create Java classes and objects, etc.). The overhead associated with the Java object accesses is considered acceptable because in most cases the external library conducts “nontrivial tasks that overshadow the overhead of this interface” [26]. This is not true of tracepoints, which are meant to quickly and efficiently commit values (the arguments of the tracepoint call) to the LTTng event streams. On the other hand, a tracepoint provider is not expected to access Java objects beyond those passed in as arguments, so the concern is rather moot.

Primitive Java types (`boolean, byte, short, int, long, char, float, double`; see [29] section 4.2: *Primitive Types and Values* and [30]) map directly to C equivalents (respectively `char, signed char, signed short, signed long, signed long long, unsigned short, float, double`), so no JNI accesses are required. Arrays of primitives will suffer from some overhead, as will the ever-troublesome strings. As a type becomes more complex (structs, unions, etc.), so will the run-time overhead increase. Since tracepoints only very rarely have complex types among their arguments, performance concerns should not be much of an issue.

It looks like the answer to the earlier question is yes, one could in theory write a Java object and its supporting C library which, given a library name, would load the library and define a Java class for each tracepoint provider found in the library. Invoking a tracepoint would then boil down to something like:

```
JavaTracepointProvider jtp = new JavaTracepointProvider(  
    library_name, tracepoint_provider_name);  
jtp.tracepoint(event_name, object_array);
```

Intermediate factory objects would bind to the `library_name` and then to the `tracepoint_provider_name` so the overhead of loading a library, scanning it for tracepoint providers, scanning them for events, and generating the Java tracepoint event classes would only be encountered once. Alternate methods could be supplied for common event payloads (e.g. no payload, single primitive type, etc.) to avoid the per-invocation overhead of manipulating Java Object arrays.

4.4.2 liblttng-ust-java

In the `lttng-ust` package, there is a `liblttng-ust-java` subfolder which gets compiled and installed if Java is detected (see Section 3.3.3.3 and Topical Note A.9). It installs a `liblttng-ust-java.so` shared library, written in C, which acts as a container for a particular tracepoint provider, `lttng_ust_java`. From Java, an `org.lttng.ust.LTTngUst` object (packaged in `liblttng-ust-java.jar`) wraps native calls to `liblttng-ust-java.so`.

`LTTngUst` is an abstract class with a handful of static methods; you call `LTTngUst.init()` to connect to the native library (this is analogous to using `dlopen` for the same purpose) and later invoke the tracepoint methods as needed, depending on the payloads:

```
public static native void tracepointInt(String name,  
    int payload);  
public static native void tracepointIntInt(String name,  
    int payload1, int payload2);  
public static native void tracepointLong(String name,  
    long payload);  
public static native void tracepointLongLong(String name,  
    long payload1, long payload2);  
public static native void tracepointString(String name,  
    String payload);
```

Other types of payloads (e.g. an integer and a long, etc.) could readily be added to the native library and `LTTngUst` by expanding their code and recompiling.

The trace events generated by this library are named `<type>_event`, where `<type>` identifies the payload (e.g. `int_event`, `int_int_event`, etc.). The `name` parameter of each method is stored as a payload prefixed element, the `name` field. For example, the `LTTngUst.tracepointInt` method generates an `int_event` with a two-fields payload of `name` and `int_payload`.

LTTngUst does not protect access to its native tracepoint methods: incorrect Java code could readily try to invoke them before invoking `LTTngUst.init()`, which would cause a `java.lang.UnsatisfiedLinkError` exception. The `LTTngUst.init()` method should have been made part of the class's static initializer; with appropriate exception handling, it would accommodate an absent `liblttng-ust-java.so` by rendering the tracepoint methods inoperative. This would mimic the C behaviour quite nicely.

LTTngUst does not have an `unload()` method that would be the analogue of C's `dlclose()`. The problem is that Java's `System` class has no `unloadLibrary` method. One possible solution would be to use *two* external libraries: a version of `libtp.so` that exposes clear entry points to its tracepoints, and a `libtpJNI.so` "handler" that would take care of `dlopening` and `dlclosing` `libtp.so`, controlled from Java using JNI.

4.4.3 Another Java tracepoint provider example

The DVD-ROM's `lttng-ust` package contains, in its `/usr/src/lttng-ust/doc/examples/drdc` folder, a `dynamic_java` example.

In this example, `tp.c` (see Section 4.1.2.1) is the starting point. It turns out the simplest approach is to work from Java towards C. First, one must choose where to package the Java object that will wrap the C shared library (here named `liblttngtp.so`). The package will be `org.lttng.ust.drdc.tpJava`:

```
package org.lttng.ust.drdc;

public abstract class tpJava {
    private static boolean loaded = false;

    static {
        load();
    }

    public static boolean isLoaded() {
        return loaded;
    }

    /**
     * Load the tracepoint provider.
     *
     */
    public static boolean load() {
        if (loaded) return true;
        try {
            System.loadLibrary("tpJNI");
            loaded = true;
        } catch (SecurityException se) {
            System.err.println(se.toString());
        } catch (UnsatisfiedLinkError ule) {
            System.err.println(ule.toString());
        } catch (NullPointerException npe) { //Can't happen
            System.err.println(npe.toString());
        }
        return loaded;
    }

    /**
     * Safely insert a sample_component:event tracepoint.
     * @param text The payload.
     */
    public static void tracepoint(String text) {
        if (loaded) _tracepoint(text);
    }

    /**
     * Insert a sample_component:event tracepoint.
     * @param text The payload.
     */
    private static native void _tracepoint(String text);
}
```

Note that the `System.loadLibrary` call converts its argument according to platform-specific procedures. On Linux, `System.loadLibrary("tpJNI")` is equivalent to something like `gcc -ltpJNI`, which will look for `libtpJNI.so`. The static initializer attempts to load the external library, so the static class is immediately usable. The native method that actually invokes the tracepoint (`_tracepoint`) is private so the public access to the tracepoint (`tracepoint`) can fail gracefully if the library could not be loaded. In this way the Java application behaves just like the C application.

Next, a few lines are added to `tp.c` to turn it into `tpJNI.c`. Note that the `tp.h` header file remains unmodified. The `org_lttng_ust_drdc_tpJava.h` file is automatically generated by running ‘`javah org.lttng.ust.drdc.tpJava`’ once `tpJava.java` has been compiled.

```
#include "org_lttng_ust_drdc_tpJava.h"
#define TRACEPOINT_CREATE_PROBES
#include "tp.h"
JNIEXPORT void JNICALL
Java_org_lttng_ust_drdc_tpJava__1tracepoint(
    JNIEnv *env,
    jclass jcls,
    jstring text)
{
    jboolean iscopy;
    const char *txt_cstr = (*env)->GetStringUTFChars(env, text, &iscopy);
    tracepoint(sample_component, event, txt_cstr);
    (*env)->ReleaseStringUTFChars(env, text, txt_cstr);
}
```

The `JNIEXPORT` lines of `tpJNI.c` would be much simpler if strings weren’t involved: integers and the like can be passed readily between the Java and C environments because they have the same representation. Recall that passing strings between C++ and C was also a delicate operation.

Finally, here is a small Java application that runs like the original `sample.c`:

```
package org.lttng.ust.drdc;

class sample {

    public static void main(String[] args) {
        int i = 0;

        System.out.println("sample starting");

        while (i < 10000) {
            //Prepare themessage in this module
            String themessage = String.format("Hello World %d", i);
            tpJava.tracepoint(themessage);
            try {
                Thread.sleep(0, 1000); // 0 ms + 1000 ns = 1 us
            } catch (InterruptedException ie) {
                //No action
            };
            i++;
        }
        System.out.println("sample done");
    }
}
```

If `sample.java` were in a different package than `tpJava`, you'd need to import the latter.

To recap:

```
$ javac -d . org/lttng/ust/drdc/tpJava.java
$ javah org.lttng.ust.drdc.tpJava
$ javac -d . org/lttng/ust/drdc/sample.java
```

The `org/lttng/ust/drdc` path will be different if you choose different packaging of your Java classes, of course. At this point the `JNIEXPORT` header is extracted from `org_lttng_ust_drdc_tpJava.h` and one can complete `tpJNI.c` by naming the arguments and writing the function bodies. Compilation then resumes:

```
$ gcc -I. -fPIC -I/usr/lib/jvm/java-7-openjdk-amd64/include \
      -D TRACEPOINT_DEFINE -D TRACEPOINT_PROBE_DYNAMIC_LINKAGE \
      -c -o libtpJNI.o tpJNI.c
$ gcc -shared -Wl,-soname,libtpJNI.so.1 -Wl,-no-as-needed \
      -o libtpJNI.so.1.0 -L/usr/local/lib -lltng-ust libtpJNI.o
$ ln -sf libtpJNI.so.1.0 libtpJNI.so.1
$ ln -sf libtpJNI.so.1 libtpJNI.so
```

The `/usr/lib/jvm/java-7-openjdk-amd64` path will vary depending on the particular JDK you have installed. You can jar your Java classes for distribution, of course.

4.5 Instrumenting a 32-bit application on a 64-bit system

In order to trace 32-bit applications running on a 64-bit system, LTTng uses a dedicated 32-bit consumer daemon. This section discusses how to build that daemon (which is *not* part of the default 64-bit build) and the LTTng 32-bit tracing libraries, and how to instrument a 32-bit application in that context.

First off, the 32-bit libraries of a number of packages LTTng relies on must be installed. With Ubuntu 12.04.3, this is mostly achieved by installing the `gcc-multilib` and `g++-multilib` packages. In a more general development context, the `ia32-libs` package could also be of use, as well as a scattering of `lib32*` packages (LTTng doesn't need `ia32-libs` or any `lib32*` packages). However, the 32-bit versions of `popt` and `uuid` are required, and they are unfortunately not in `ia32-libs`. The 32-bit `uuid` library is supplied by the `libuuid1:i386` package (after installation, a symbolic link from `/usr/lib/i386-linux-gnu/libuuid.so` to `/lib/i386-linux-gnu/libuuid.so.1.3.0` must be added). The 32-bit `popt` library is more elusive: this document recommends the CBLFS (Community-driven Beyond Linux From Scratch) `Popt` page (<http://cblfs.cross-lfs.org/index.php/Popt>) from which the `popt-1.16` tarball can be downloaded. Follow the page's instructions to create the 32-bit library, changing the install target from `/usr/lib` to `/usr/local/lib32`.

Next, the sources from the `userspace-rcu`, `lttng-ust` and `lttng-tools` packages should be deployed to a 32-bit working directory. Whereas the 64-bit packages were built from `/usr/src`, here each package is deployed to `~/userspace-rcu-32`, `~/lttng-ust-32` and `~/lttng-tools-32`. A partial 32-bit LTTng toolchain must be built, being careful not to overwrite the installed 64-bit toolchain.

The build procedure is as before (see Section 3.3.3), except that the `./configure` line changes.

4.5.1 userspace-rcu-32

```
$ ./bootstrap
$ ./configure --libdir=/usr/local/lib32 CFLAGS=-m32
$ make
$ sudo make install
$ sudo ldconfig
```

To capture logs of each step, each command line can be suffixed in one of two ways:

```
$ ./bootstrap &> bootstrap.log
$ ./bootstrap 2>&1 | tee bootstrap.log
```

The first method captures all output to the log file; the second does the same but simultaneously lets the output appear in the console.

The key change here is that the resulting 32-bit libraries are installed in `/usr/local/lib32` instead of the usual `/usr/local/lib`. This prevents the 64-bit libraries from being overwritten.

4.5.2 lttng-ust-32

```
$ ./bootstrap  
$ ./configure --libdir=/usr/local/lib32 CFLAGS=-m32 \  
  LDFLAGS=-L/usr/local/lib32  
$ make  
$ sudo make install  
$ sudo ldconfig
```

The `LDFLAGS` is required in order for the build to find the 32-bit `userspace-rcu` libraries it requires.

If Java and/or SystemTap are installed, options may be added to the `./configure` line as explained in Section 3.3.3.3.

4.5.3 lttng-tools-32

```
$ ./bootstrap  
$ ./configure --libdir=/usr/local/lib32 CFLAGS=-m32 \  
  LDFLAGS=-L/usr/local/lib32 --bindir=/usr/local/bin32 \  
  --with-consumerd-only \  
  --with-consumerd64-libdir=/usr/local/lib \  
  --with-consumerd64-bin=/usr/local/lib/lttng/libexec/lttng-consumerd  
$ make  
$ sudo make install  
$ sudo ldconfig
```

The `with-consumerd-only` option requests that only `lttng-consumerd` be built. The `with-consumerd64-bin` and `with-consumerd64-libdir` options ensure the 32-bit `lttng-tools` components find the 64-bit consumer daemon. The `with-consumerd-only` make currently fails because it tries to make too many tests [31]; this can be fixed by editing the `Makefile` and removing the `tests` sub-directory from the `SUBDIRS` variable (line 305) before running `configure`. The suppressed tests are not required in order to have a functioning 32-bit LTTng toolchain.

The `bindir` configure option is given only for completeness: it is not actually used nor created by the compilation.

This completes the 32-bit LTTng toolchain. All that is left to do is to redo the 64-bit `lttng-tools` to hook in the 32-bit `lttng-consumerd`.

4.5.4 lttng-tools-64 (/usr/src/lttng-tools)

```
$ ./bootstrap
$ ./configure LDFLAGS=-L/usr/local/lib \
--with-consumerd32-libdir=/usr/local/lib32 \
--with-consumerd32-bin=/usr/local/lib32/lttng/libexec/lttng-consumerd
$ make
$ sudo make install
$ sudo ldconfig
```

Henceforth, the `lttng-sessiond` daemon will automatically find and use the 32-bit `lttng-consumerd` if required.

4.5.5 Making a source of 32-bit user-space events

The `lttng-ust/doc/examples/drdc` directory of samples includes a `static32` target in its `Makefile`. This target is not built automatically.

```
static32: static32.o tp32.o
@echo "~~~~~Linking sample_@"
$(CC) -m32 -o sample_$@ $^ $(LDFLAGS) $(LIBDL) $(LIBUST) \
-Wl,-rpath,'/usr/local/lib32',--enable-new-dtags
@echo "    Use './sample_$@' to run sample_$@"

static32.o: sample.c tp.h
@echo "~~~~~Compiling $@"
$(CC) -m32 $(CPPFLAGS) $(LOCAL_CPPFLAGS) $(TP_DEFINE) -c -o $@ $<

tp32.o: tp.c tp.h
@echo "~~~~~Compiling $@"
$(CC) -m32 $(CPPFLAGS) $(LOCAL_CPPFLAGS) -c -o $@ $<
```

The target will look for the 32-bit `liblttng-ust` library in `/usr/local/lib32` first (that's what the `rpath` linker tag does). To build this 32-bit user-space event generator (once the 32-bit LTTng toolchain is in place), open a console in `lttng-ust/doc/examples/drdc`, and invoke:

```
$ LDFLAGS=-L/usr/local/lib32 make static32
```

LTTng can capture 32- and 64-bit user-space events with this simple script:

```
$ lttng create session3264
$ lttng enable-event -u -a
$ lttng start
$ ./sample_static
$ ./sample_static32
$ lttng destroy
```

This will create a trace named `session3264` in `~/lttng-traces` with both a `/ust/uid/1000/32-bit` component and a `ust/uid/1000/64-bit` component.

4.6 Instrumenting the kernel space

If the `lttng-tools` package is installed, `linux-headers` will already be present because `lttng-tools` depends on it. This is sufficient if the intent is to add a new instrumented module to the kernel (see Section 4.6.6). If the intent is to instrument existing kernel modules or the kernel itself, then the appropriate `linux-source` package is needed.

Tracepoints are already integrated into the kernel (since version 2.6.28, December 2008). For example, the `sched_switch` event is traced in the kernel's `kernel/sched.c` `prepare_task_switch` function (for the 3.2 kernel), which calls `trace_sched_switch` last:

```
static inline void
prepare_task_switch(struct rq *rq,
                    struct task_struct *prev,
                    struct task_struct *next)
{
    sched_info_switch(prev, next);
    perf_event_task_sched_out(prev, next);
    fire_sched_out_preempt_notifiers(prev, next);
    prepare_lock_switch(rq, next);
    prepare_arch_switch(next);
    trace_sched_switch(prev, next);
}
```

Tracepoints can generally be counted on to remain forward-compatible as the kernel evolves, although the precise timing may sometimes be altered. For instance, the `prepare_task_switch` function for the 3.9 kernel is in `kernel/sched/core.c` and calls `trace_sched_switch` first:

```
static inline void
prepare_task_switch(struct rq *rq,
                    struct task_struct *prev,
                    struct task_struct *next)
{
    trace_sched_switch(prev, next);
    sched_info_switch(prev, next);
    perf_event_task_sched_out(prev, next);
    fire_sched_out_preempt_notifiers(prev, next);
    prepare_lock_switch(rq, next);
    prepare_arch_switch(next);
}
```

As explained in `include/linux/tracepoint.h` (found in either of the `linux-headers` and `linux-source` package installations), the `trace_sched_switch` call is realised by a `TRACE_EVENT` macro. *There is no explicit declaration of the `trace_sched_switch()` function.* It only appears as `TRACE_EVENT(sched_switch...)`, in the `trace/events/sched.h` of the kernel’s scheduler module (`kernel/sched.c` or `kernel/sched/core.c`, found in the `linux-source` package installation). All kernel tracepoints headers can thus be located by searching for `TRACE_EVENT` in the kernel source code. The event names defined in these headers yield the `trace_*` function names one needs to search for within the kernel source code⁹. By contrast, user-space instrumentation (see Section 4.1.2 above) “anonymizes” the tracepoint invocations by using `tracepoint()` calls which are realised by `TRACEPOINT_EVENT` macros.

The tracepoint providers for the kernel are the LTTng kernel modules: until they are loaded, the kernel instrumentation remains inert. The LTTng kernel modules are loaded by the `lttng-sessiond` daemon when it is spawned (the list of modules the daemon may load is found in `lttng-tools/src/bin/lttng-sessiond/modprobe.c`; see also Section 2.1.7 and 3.3.4), and the first thing each module does is figure out where the tracepoints it manages are in the kernel and its modules. Kernel modules have a `.ko` extension and reside in `/lib/modules/<kernel_version_name>/`, mostly in the `kernel` sub-directory; they can be loaded and unloaded from the kernel on demand. They offer an easy way to dynamically adjust the functionality of the base kernel without having to rebuild or recompile it. Most device drivers are implemented as kernel modules for this reason.

The next sections will discuss LTTng kernel tracepoint headers, LTTng tracepoint provider kernel modules, custom kernel modules, custom kernels, and finally how to instrument either the kernel or one of its modules. The latter process is described in extremely succinct terms by `lttng-modules/instrumentation/events/README`.

4.6.1 The kernel tracepoint header

Kernel tracepoint headers are broadly similar to user-space tracepoint headers (see Section 4.1.2.2), but with important differences.

While describing the tracepoints is done using a variety of macros (see Sections 4.6.1.3 through 4.6.1.5), a design decision must also be made: should the new tracepoints be added to an existing LTTng kernel tracing “system” (see below), or should they form a wholly new LTTng kernel tracing “system”? The choice is of no consequence when writing the tracepoint macros unless a new tracepoint relies on an existing tracepoint template (see `DECLARE_EVENT_CLASS` in Section 4.6.1.3): it matters only in terms of where the new tracepoint declarations should be written.

⁹ None of the `trace_*` functions are exposed as kernel symbols, with the single exception of the `rcu_utilization` event.

The set of LTTng kernel tracepoint definitions is broken down into *systems*, each “system” being handled by a separate LTTng kernel tracepoint provider (a separate LTTng kernel module). The current LTTng “systems” are: asoc, block, btrfs, compaction, ext3, ext4, gpio, irq, jbd, jbd2, kmem, kvm, kvmmmu, lock, lockdep, lttnng_statedump, module, napi, net, power, printk, random, raw_syscalls, rcu, regmap, regulator, rpm, sched, scsi, signal, skb, sock, sunrpc, timer, udp, vmscan, workqueue, and writeback. Obviously, these names cannot be used for new “systems”.

If the choice is to extend an existing LTTng kernel tracing “system”, the macros are simply added to the appropriate `<system>.h` file. If the choice is to create a new “system”, then a new `<system>.h` file must be created.

4.6.1.1 Dual nature of the kernel tracepoint headers

LTTng, when considered as a package of files, faces a continuing problem: adapting to multiple kernel configuration options and evolutionary changes. Over time, the Linux kernel’s internal structure changes as facilities are added and removed. Despite a strong commitment to “breaking” the programming interface of the kernel as little as possible on the part of the kernel designers, it does happen occasionally and unavoidably. The internals of the kernel are even more susceptible to significant changes as they are not bound by an interface contract. The end result is that the set of tracepoints changes over time, and the signatures of some of them also change over time. The `linux-source` and `linux-headers` packages readily deal with this by changing their `include/trace/events` headers as needed. By contrast, LTTng’s `lttng-modules` package has to be buildable against any kernel version since 2.6.28 (there were no kernel tracepoints at all before 2.6.28).

LTTng solves this problem by supplying its own set of tracepoint headers (in `lttng-modules/instrumentation/events/lttng-module`). These use conditional preprocessing directives to dynamically adapt to the particular kernel LTTng is being built against, in terms of both its configuration and options. For instance, here are a few lines extracted from the `lttng-module` version of `power.h` (the conditional preprocessing directives are highlighted in yellow):

```
#ifdef CONFIG_EVENT_POWER_TRACING_DEPRECATED

/*
 * The power events are used for cpuidle & suspend (power_start,
 * power_end) and for cpufreq (power_frequency)
 */
DECLARE EVENT CLASS(power,
#ifndef LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,36))
    TP_PROTO(unsigned int type, unsigned int state,
             unsigned int cpu_id),
    TP_ARGS(type, state, cpu_id),
#else
    TP_PROTO(unsigned int type, unsigned int state),
    TP_ARGS(type, state),
#endif
[...]
```

In the example above, the `power` event template (see Section 4.6.1.3 below) is defined only if the kernel configuration includes the `CONFIG_EVENT_POWER_TRACING_DEPRECATED` flag. Further, the tracepoint's signature grows by an additional argument with kernel version 2.6.36. This change in the `TP_PROTO` and `TP_ARGS` macros is also later reflected in the event template's `TP_STRUCT_entry`, `TP_fast_assign` and `TP_printk` macros (not shown here).

Both `<system>.h` copies are needed to build the corresponding LTTng tracepoint provider kernel module, while all other executables (instrumented kernel units or modules) only require the `linux-headers` copy. So far, this duplication of tracepoint headers has only one implication when writing a new kernel tracepoint header: the new `<system>.h` file will need to be copied to both locations, `linux-headers/include/trace/events` (the mainline location) and `lttng-modules/instrumentation/events/lttng-module` (the `lttng-module` location).

However, LTTng takes advantage of the existence of a separate `lttng-module` directory to define a number of helper macros. This has the consequence that the syntax of the `lttng-module` copies of the tracepoint headers is slightly different. The differences are explained in Sections 4.6.1.3 through 4.6.1.5. Thus, each new `<system>.h` file must actually be written twice. This document preconizes the use of `#ifdef` statements in order to write both files at once, a practice which minimises the danger of discrepancies creeping in between the two flavours (see the example in Section 4.6.6). This is shown in the top part of Figure 35.

A third copy of the `<system>.h` file, identical to that found in the `linux-headers` of whichever kernel version is current at the time of its writing, should be put in the `/usr/src/lttng-modules/instrumentation/events/mainline` directory for reference. It can be used to determine if adjustments are required when the module is built against a different release of the kernel. The `mainline` directory is not used for the compilation of new LTTng tracepoint systems or custom modules and may be omitted entirely. It appears in grey in Figure 35.

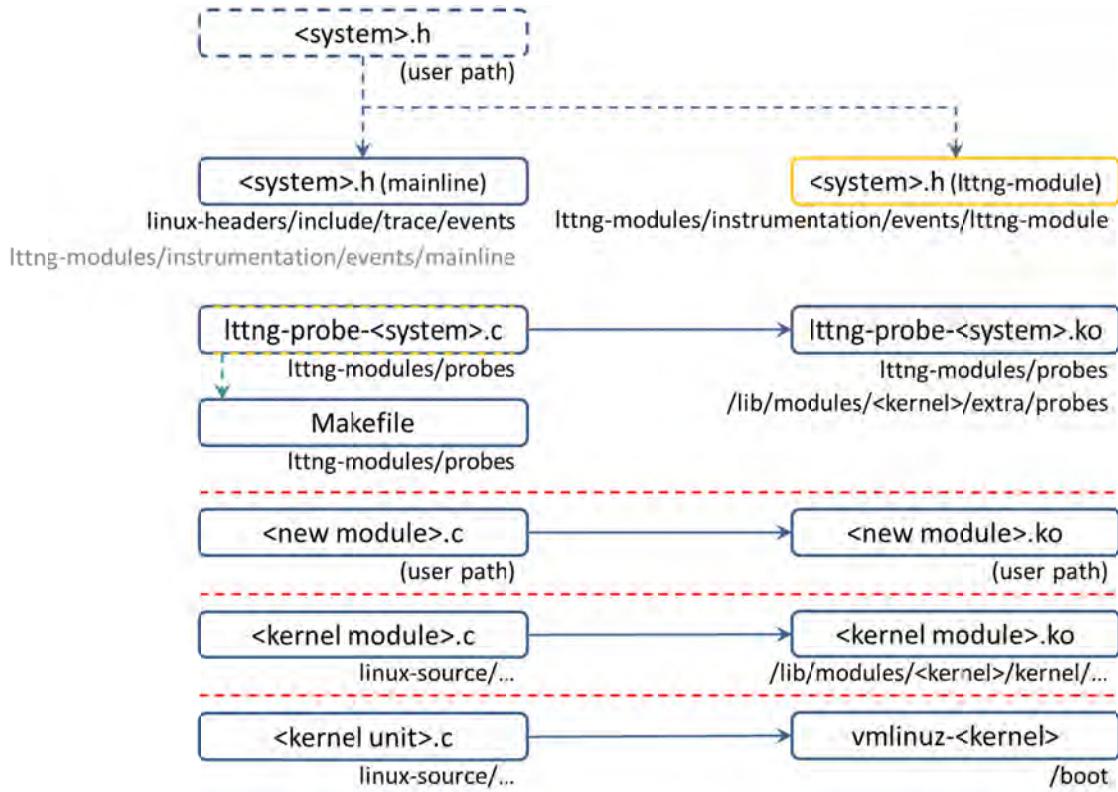


Figure 35: Relationships between source files and objects when instrumenting the kernel. The LTTng kernel module (`lttng-probe-<system>`) is the only unit whose build depends on the lttng-module version of the `<system>.h` file; this dependency is shown in yellow.

To summarise, when extending an existing LTTng kernel tracing “system”, two `<system>.h` files must be edited. When creating a new “system”, a dual-use `<system>.h` file is created (shown dashed in Figure 35). Further, in the latter case a line must be added to the `lttng-modules/probes/Makefile` to incorporate the new tracepoint provider in the build of `lttng-modules`.

The LTTng kernel module source files are called `lttng-probe-<system>.c` (they are very simple wrappers around the `<system>.h` files) and generate `lttng-probe-<system>.ko` modules. Once the new tracepoint provider (or the extended tracepoint provider) is built, installed and loaded, LTTng is able to trace the new tracepoint (see Section 4.6.3).

The lower part of Figure 35 shows the possible instrumentations: a new kernel module may be created and instrumented, an existing kernel module may be instrumented, or the kernel itself may be instrumented. The instrumentation process is much the same in each case (see Sections 4.6.4 through 4.6.6), and the three cases may be combined freely (i.e. it is possible to insert the new tracepoint in all three locations).

4.6.1.2 A kernel tracepoint template

When creating a new “system”, the kernel tracepoint macros should be written in a header template like the following one (hello.h; the template is also found in `lttng-ust/doc/examples/drdc/kernel`). The template’s `hello` is a placeholder for the name of the new LTTng tracepoint system. The template parts highlighted in yellow must be filled in as appropriate:

```
//These two #defines handle the subtle differences between
//the lttng-module and mainline versions of this file.
//For mainline:
#define _LTTNG_MAINLINE
#define _LTTNG_SEPARATOR ;
//For lttng-module:
//#undef _LTTNG_MAINLINE
//#define _LTTNG_SEPARATOR /* */

//The LTTng tracepoint provider kernel module will be
//named 'lttng-probe-TRACE_SYSTEM'
#undef TRACE_SYSTEM
#define TRACE_SYSTEM hello
//If this header is not named TRACE_SYSTEM.h,
//then specify the name here:
//#define TRACE_INCLUDE_FILE hello
//This header is to be copied to linux-source's
//(or linux-header's) include/TRACE_INCLUDE_PATH
//#define TRACE_INCLUDE_PATH trace/events

#if !defined(_TRACE_HELLO_H) || defined(TRACE_HEADER_MULTI_READ)
#define _TRACE_HELLO_H

#include <linux/tracepoint.h>

#ifndef _TRACE_HELLO_DEF
#define _TRACE_HELLO_DEF
//Declare consts, enums, working structs, helper functions here
#endif /* _TRACE_HELLO_DEF */

//Use TRACE_EVENT, DECLARE_EVENT_CLASS and DEFINE_EVENT
//to define tracepoints here.

#endif /* _TRACE_HELLO_H */

/* This part must be outside protection */
#undef _LTTNG_SEPARATOR
#ifdef _LTTNG_MAINLINE
#include <trace/define_trace.h>
#else
#include "../../../probes/define_trace.h"
```

```
#endif
```

The template's `_LTTNG_MAINLINE` and `_LTTNG_SEPARATOR` defines are used to let a single kernel tracepoint header file be used in both the `linux-headers` and `lttng-module` contexts, as will be shown later.

The optional `TRACE_INCLUDE_PATH` define makes it possible to place the header elsewhere than `include/trace/events` in the `linux-headers`. The optional `TRACE_INCLUDE_FILE` define makes it possible to name the header differently from the resulting kernel tracepoint provider module (e.g. `TRACE_SYSTEM hello` and `TRACE_INCLUDE_FILE goodbye` would mean a `goodbye.h` header and `lttng-probe-hello.c` LTTng tracepoint provider kernel module). You *must* define `TRACE_SYSTEM`, which sets the LTTng kernel tracepoint provider module name. It is expected that the tracepoint names will begin with the `TRACE_SYSTEM` value.

The following sub-sections discuss the various macros used in defining kernel tracepoints. They are necessary when modifying an extant kernel tracepoint header or creating a new one.

4.6.1.3 The `TRACE_EVENT` and `DECLARE_EVENT_CLASS` macros

Each `TRACE_EVENT` macro defines a single tracepoint. Each `DECLARE_EVENT_CLASS` macro defines an abstract tracepoint, a *template* which is used by any number of later `DEFINE_EVENT` macros to define actual tracepoints. Using `DECLARE_EVENT_CLASS` and `DEFINE_EVENT` makes it possible to succinctly define families of tracepoints that are identical except for their names. A number of variants exist for each of these three macros; they are explained later on.

Unlike user-space tracepoints, which have a `TRACEPOINT_LOGLEVEL` macro to set the logging level of the event (see Section 4.1.2.4), LTTng kernel tracepoints do not yet have any means of setting the level to anything else than the default, `TRACE_EMERG`.

The `TRACE_EVENT` and `DECLARE_EVENT_CLASS` macros are written in precisely the same way, as in this example:

```
TRACE_EVENT(<event name>,
    //The tracepoint function's argument signature
    TP_PROTO(unsigned int ret, char *str, int *intarr, int arrsize),
    //Names for the arguments, for use in TP_STRUCT, etc.
    TP_ARGS(ret, str, intarr, arrsize),
    //The tracepoint payload field definitions and names
    TP_STRUCT_entry(
        __field(unsigned int, return)
        __string(s1, str)
        __array(int, array, arrsize)
        __dynamic_array(char, s2, strlen(str)*sizeof(char)))
    ),
    //The payload setting instructions
    TP_fast_assign(
        __entry->return = ret;
        __assign_str(s1, str);
        memcpy(__entry->array, intarr, arrsize*sizeof(int));
        memcpy(__get_dynamic_array(s2), str, strlen(str)*sizeof(char)));
    ),
    //The kernel message buffer printout (optional)
    TP_printk(
        "event_name ret=%d str=%s", __entry->return, __get_str(s1)
    )
);
```

The example above is for ‘mainline’ (the Linux headers); the `lttng-module` version differs in two mandatory ways. First, instead of ending with a parenthesis-semi-colon (‘`;’’), it ends with a lone parenthesis (‘)’’). This is handled by the _LTTNG_SEPARATOR define. Second, the arguments of TP_fast_assign, instead of being a series of semi-colon-separated executable instructions, are a series of space-separated macro invocations. This is explained in the TP_fast_assign section below (4.6.1.3.4).`

The `<event name>` appearing in `TRACE_EVENT`, `DECLARE_EVENT_CLASS` or `DEFINE_EVENT` is an identifier. The tracepoint will be invoked by the `trace_<event name>()` function.

4.6.1.3.1 TP_PROTO

The `TP_PROTO` macro defines the `trace_<event name>` function's signature, that is to say, the types of its arguments. It is written as a standard C function signature: a comma-separated list of type-name pairs (known as the “function declarator's parameter type list” in [23]). Below are some examples:

```
TP_PROTO(struct snd_codec *codec, unsigned int reg),  
TP_PROTO(const char *name),  
TP_PROTO(struct request_queue *q, int prio, bool explicit),  
TP_PROTO(struct bio *bio, dev_t dev, sector_t from),  
TP_PROTO(struct page *page, u64 start, unsigned long date),
```

4.6.1.3.2 TP_ARGS

The `TP_ARGS` macro reprises the identifiers of the arguments (this is the “function declarator's identifier list” in [23]). The identifiers *should* match exactly.¹⁰ The `TP_PROTO` declarations of the previous example would yield these corresponding `TP_ARGS` declarations:

```
TP_ARGS(codec, reg),  
TP_ARGS(name),  
TP_ARGS(q, prio, explicit),  
TP_ARGS(bio, dev, from),  
TP_ARGS(page, start, date),
```

4.6.1.3.3 TP_STRUCT_entry

The `TP_STRUCT_entry` macro defines the payload fields using four different macros according to the field type:

```
__field(<type>, <field>)  
__array(<type>, <field>, <size>)  
__dynamic_array(<type>, <field>, <size>)  
__string(<field>, <string>)
```

In each case, `<type>` is a scalar type declaration (e.g. `struct snd_codec` or `unsigned int` or `bool`, etc.). The `<field>` is the field's identifier. For arrays, `<size>` is an expression yielding an integer, the number of occurrences of the `<type>`. Multi-dimensional arrays should be unrolled. Strings are simply dynamic arrays of `char` (with the additional constraint that the last `char` is `\0`), and you are free to treat them as such. The `__string` declaration will use `strlen()` on its `<string>` argument to dynamically figure out the field's size.

¹⁰ `TP_ARGS` translates identifiers used by later instructions into an index into the tracing function's argument array. Compilation won't detect the (erroneous) case where the identifiers are reprise in a different order unless a type incompatibility arises. For example, if one has:

```
TRACE_EVENT(event, TP_PROTO(int a, int b), TP_ARGS(b, a),  
TP_STRUCT_entry(__field(int, fielda)),  
TP_fast_assign (__entry->fielda = a;))
```

Then a call to `trace_event(1, 2)` will store 2 in the `fielda` event record field, not 1.

Optionally, the `lttng-module` version of the kernel tracepoint header may use suffixed versions of `_field`, `_array`, and `_dynamic_array` to provide hints to the trace metadata. These are used by `babeltrace` to alter the event record textual representation:

- ◆ `_field_network` indicates the field's bytes use network ordering;
- ◆ `_field_hex` indicates the field value should be rendered in hexadecimal notation;
- ◆ `_field_network_hex` combines the two;
- ◆ `_array_text` and `_dynamic_array_text` indicate the array of characters is UTF-8-encoded; and
- ◆ `_array_hex` and `_dynamic_array_hex` indicate the array member values should be rendered in hexadecimal notation.

Using `TP_STRUCT__entry()`, a tracepoint can be made payload-less. The LTTng tracepoint handler kernel module compiles without complaint, but compilation of the instrumented module causes a couple of warnings:

```
make[1]: Entering directory "/usr/src/linux-headers-3.2.0-53-generic"
  CC [M] /home/user/Documents/mymodule/hello.o
In file included from include/trace/ftrace.h:356:0,
                  from include/trace/define_trace.h:86,
                  from include/trace/events/hello.h:151,
                  from /home/daniel/Documents/mymodule/hello.c:5:
include/trace/events/hello.h: In function
  'ftrace_define_fields_hello_nil':
include/trace/events/hello.h:130:1: attention : unused variable
  'field' [-Wunused-variable]
include/trace/events/hello.h:130:1: attention : 'ret' is used
  uninitialized in this function [-Wuninitialized]
Building modules, stage 2.
MODPOST 1 modules
  LD [M] /home/user/Documents/mymodule/hello.ko
make[1]: Leaving directory "/usr/src/linux-headers-3.2.0-53-generic"
```

4.6.1.3.4 `TP_fast_assign`

The `TP_fast_assign` macro specifies how the payload fields are to be set from the arguments. In the `linux-headers` (mainline) version of the header, these are regular C statements, separated by semi-colons. The payload structure instance is identified as `_entry`. The sequence in which the statements appear need not match the `TP_STRUCT__entry` sequence of field declarations, although this is good programming practice. The declarations below continue the `TP_STRUCT__entry` examples:

```
_entry-><field> = <expr>;
memcpy(_entry-><field>, <expr>, <size>*sizeof(<type>));
memcpy(_get_dynamic_array(<field>), <expr>, <size>*sizeof(<type>));
__assign_str(<field>, <expr>);
```

The fields of `__entry` can be referred to directly only in the case of scalar fields and static arrays; for dynamic arrays and strings, the `__get_dynamic_array` and `__get_str(<field>)` functions must be used, respectively (`__get_str` is mostly useful within `TP_printk`, below). Strings fields can be written to using the `__assign_str` function. Finally, `__get_dynamic_array_len(<field>)` is also available. Note also that the `memcpy` standard function expects a `<size>` in *bytes*, unlike the `TP_STRUCT__entry` declarations which expect `<size>` in *number of elements*.

In the `lttng-module` version of the header, the `TP_fast_assign` statements must be rewritten using particular space-separated macros. The declarations given above would become:

```
tp_assign(<field>, <expr>)
tp_memcpy(<field>, <expr>, <size>*sizeof(<type>))
tp_memcpy_dyn(<field>, <expr>)
tp_strcpy(<field>, <expr>)
```

Be careful with `tp_memcpy_dyn`: it relies on the metadata set up by the `TP_STRUCT__entry` part of the macro. If in doubt, you can use `tp_memcpy` instead.

4.6.1.3.5 `TP_printk`

The `TP_printk` macro, finally, specifies how the event is to be recorded in the kernel's log buffer (this is used by ftrace plugins that make use of the tracepoints). It follows the same rules as the standard `printk` function: a format string followed by a number of arguments (`printk` behaves similarly to `printf` with a few minor differences, the main one being lack of floating-point support). Here are some sample `TP_printk` macros:

```
TP_printk("%llu + %d", (unsigned long long) __entry->s, __entry->err)
TP_printk("%s", "Arbitrary string")
TP_printk("%s", "") //Still prints a \n
```

4.6.1.4 The TRACE_EVENT_CONDITION and TRACE_EVENT_MAP macros

The TRACE_EVENT_CONDITION macro is a variant of TRACE_EVENT that inserts an argument in fourth position (between TP_ARGS and TP_STRUCT_entry). This argument uses TP_CONDITION(<exp>) and evaluates the Boolean expression <exp> (typically involving some of the TP_ARGS). If the TP_CONDITION is false, the event is not emitted. For example:

```
TRACE_EVENT_CONDITION(<event name>,
    TP_PROTO(const char *astring, unsigned start, unsigned end),
    TP_ARGS(astring, start, end),
    TP_CONDITION(start != end),
    TP_STRUCT_entry(
        __string(astring, astring)
    ),
    TP_fast_assign(
        __assign_str(astring, astring);
    ),
    TP_printk(
        "%s", __get_str(astring)
    )
)
```

The TRACE_EVENT_CONDITION macro is available to mainline and lttng-module.

The TRACE_EVENT_MAP macro is a variant of TRACE_EVENT that inserts an argument in second position (between <event name> and TP_PROTO). This <lttng_event_name> argument is the event name that will be used by LTTng, made distinct from the event name as used to define the kernel invocation function. This macro is used mostly to enforce naming conventions on the LTTng side of legacy kernel functionalities. For example:

```
TRACE_EVENT_MAP(machine_suspend, power_machine_suspend,
    TP_PROTO(unsigned int state),
    TP_ARGS(state),
    TP_STRUCT_entry(
        __field(u32, state)
    ),
    TP_fast_assign(
        tp_assign(state, state)
    ),
    TP_printk(
        "state=%lu", (unsigned long) __entry->state
    )
)
```

This tracepoint is named machine_suspend by the kernel and is invoked by trace_machine_suspend(). LTTng, on the other hand, names the event power_machine_suspend. The TRACE_EVENT_MAP macro is *only* available to the lttng-module version of the header.

There is also a `TRACE_EVENT_CONDITION_MAP` macro that combines `TRACE_EVENT_CONDITION` with `TRACE_EVENT_MAP`:

```
TRACE_EVENT_CONDITION_MAP(<kernel event name>, <lttng event
name>,
    TP_PROTO(...),
    TP_ARGS(...),
    TP_CONDITION(...),
    TP_STRUCT_entry(...),
    TP_fast_assign(...),
    TP_printk(...)
)
```

The `TRACE_EVENT_CONDITION_MAP` macro is also only available to the `lttng-module` version of the header.

Finally, `linux-headers/include/trace/ftrace.h` defines also the `TRACE_EVENT_FN`, `TRACE_EVENT_FN_MAP` and `TRACE_EVENT_FLAGS` macros for use with system call tracing (see `linux-headers/include/trace/events/syscalls.h`). The first two add callbacks to `TRACE_EVENT` in order to set and clear the `TIF_SYSCALL_TRACEPOINT` flags of all non-kernel threads that are undergoing system call tracing (see `linux-source/kernel/tracepoint.c`). The last one is an initializer fragment that sets the `flags` field of `ftrace_event_call` structures (see `linux-headers/include/linux/ftrace_event.h`). These macros are mentioned here for completeness's sake only, as they are unlikely to be of use outside of the specific context of system call tracing—which is handled by the special LTTng module `lttng-tracer.ko`.

4.6.1.5 The `DEFINE_EVENT`, `DEFINE_EVENT_PRINT`, `DEFINE_EVENT_CONDITION` and `DEFINE_EVENT_MAP` macros

The `DEFINE_EVENT` macro has only four arguments. The first is the `<template name>`, indicating the previous `DECLARE_EVENT_CLASS` template to use. The second is the `<event name>`. The third and fourth must repeat the template's `TP_PROTO` and `TP_ARGS`. See the example below.

The `DEFINE_EVENT_PRINT` macro is a variant of `DEFINE_EVENT` that adds a fifth argument, a replacement for the template's `TP_printk`.

The `DEFINE_EVENT_CONDITION` macro is a variant of `DEFINE_EVENT` that adds a fifth argument, a `TP_CONDITION` (see 4.6.1.4).

The `DEFINE_EVENT_PRINT` and `DEFINE_EVENT_CONDITION` macros are available to both versions (mainline and `lttng-module`) of the header.

The `DEFINE_EVENT_MAP` macro is a variant of `DEFINE_EVENT` that inserts an argument in third position (between `<event name>` and `TP_PROTO`). This `<lttng event name>` argument is the event name that will be used by LTTng, made distinct from the event name as used to define the kernel invocation function. This is used mostly to enforce naming conventions on the LTTng side of legacy kernel functionalities. For example:

```
DECLARE_EVENT_CLASS(kmem_alloc,
    TP_PROTO(...),
    TP_ARGS(...),
    TP_STRUCT_entry(...),
    TP_fast_assign(...),
    TP_printk(...)
)

DEFINE_EVENT(kmem_alloc, kmem_cache_alloc,
    TP_PROTO(...),
    TP_ARGS(...)
)

DEFINE_EVENT_PRINT(kmem_alloc, kmem_cache_alloc_var,
    TP_PROTO(...),
    TP_ARGS(...),
    TP_printk(...)
)

DEFINE_EVENT_CONDITION(kmem_alloc, kmem_cache_maybe,
    TP_PROTO(...),
    TP_ARGS(...),
    TP_CONDITION(...)
)

DEFINE_EVENT_MAP(kmem_alloc, kmalloc, kmem_kmalloc,
    TP_PROTO(...),
    TP_ARGS(...)
)
```

In the example above, `kmem_alloc` is a tracepoint template. The following `DEFINE_EVENT*` macros define several tracepoints based on the template. The first (`DEFINE_EVENT`) is named `kmem_cache_alloc` and is consequently invoked by `trace_kmem_cache_alloc()`. The second (`DEFINE_EVENT_PRINT`) is named `kmem_cache_alloc_var` and overrides the `TP_printk` of its template. The third (`DEFINE_EVENT_CONDITION`) is named `kmem_cache_maybe` and will be traced only if the `TP_CONDITION` is fulfilled. The last (`DEFINE_EVENT_MAP`) is named `kmalloc` by the kernel code and is thus invoked by `trace_kmalloc()`, but is named `kmem_kmalloc` by LTTng and must be thus named when managing tracing sessions. The `DEFINE_EVENT_MAP` macro is *only* available to the `lttng-module` version of the header.

There are also `DEFINE_EVENT_PRINT_MAP` and `DEFINE_EVENT_CONDITION_MAP` macros that combine respectively `DEFINE_EVENT_PRINT` and `DEFINE_EVENT_CONDITION` with `DEFINE_EVENT_MAP`:

```
DEFINE_EVENT_PRINT_MAP(<template>, <kernel name>, <lttng name>,
    TP_PROTO(),
    TP_ARGS(),
    TP_printk()
)
```

```
DEFINE_EVENT_CONDITION_MAP(<template>, <kernel name>, <lttng
name>,
    TP_PROTO(),
    TP_ARGS(),
    TP_CONDITION()
)
```

The `DEFINE_EVENT_PRINT_MAP` and `DEFINE_EVENT_CONDITION_MAP` macros are also only available to the lttng-module version of the header. There is as yet no `DEFINE_EVENT_PRINT_CONDITION_MAP` macro, but it could conceivably be added in the future.

4.6.2 The kernel tracepoint provider module

When new tracepoints are assigned to an existing LTTng tracepoint provider kernel module, nothing needs to be done to the corresponding source (`lttng-probe-<system>.c`): all the work is done by the header.

When creating a new LTTng kernel tracing “system”, on the other hand, a new kernel tracepoint provider (a new LTTng kernel module) must be written. Below is `/usr/src/lttng-modules/probes/lttng-probe-hello.c`. As the file is mostly boiler-plate, the varying parts are highlighted in yellow:

```
#include <linux/module.h>

/*
 * Create the tracepoint static inlines from the kernel to validate
 * that the trace event macros match the kernel we run on.
 */
#include <trace/events/hello.h>

#include "../wrapper/tracepoint.h"

/*
 * Create LTTng tracepoint probes.
 */
#define LTTNG_PACKAGE_BUILD
#define CREATE_TRACE_POINTS
#define TRACE_INCLUDE_PATH ../instrumentation/events/lttng-module

#include "../instrumentation/events/lttng-module/hello.h"

MODULE_LICENSE("GPL and additional rights");
MODULE_AUTHOR("Daniel U. Thibault <daniel.thibault@drdc-rddc.gc.ca>");
MODULE_DESCRIPTION("LTTng hello probes");
```

Important: The module must be added explicitly to `lttng-modules/probes/Makefile` before it can be compiled and installed (as part of the `lttng-modules` package). This `Makefile` isn’t really designed to be easily extended and has a complex structure of nested conditional statements. Assuming the new module’s compilation is imperative, it can simply be added alongside the `lttng-probe-statedump.o` statement near the beginning of `Makefile`:

```
[...]
obj-m += lttng-probe-statedump.o
obj-m += lttng-probe-hello.o
[...]
```

4.6.3 Installing the kernel tracepoint provider

The new kernel tracepoint provider module (or the modified LTTng kernel tracepoint provider modules) can be compiled and installed by re-installing `lttng-modules` (see Section 3.3.3.1) manually:

```
$ make &> make-hello.log  
$ sudo make modules_install &> install-hello.log  
$ sudo depmod -a
```

You *must not* re-install `lttng-modules` through Synaptic, as this will do a clean install and overwrite any modifications made to `probes/Makefile` or to any LTTng kernel tracepoint headers.

The new tracepoints will become available to LTTng kernel tracing sessions (even already-running ones) as soon as the new `lttng-probe-<system>` module is loaded. Re-installation as detailed above restarts the `lttng-sessiond` daemon, so no other action is required when modifying an existing LTTng kernel tracepoint header.

A new kernel tracepoint provider module can be added to the list of modules loaded by the kernel session daemon by editing `lttng-tools/src/bin/lttng-sessiond/modprobe.c`. The list is held by the `kern_modules_list` structure. Once `lttng-tools` is re-installed, the new kernel tracepoint provider module will be loaded and unloaded alongside the others by the root session daemon. Otherwise, manual loading is required. Here is a manual loading example (the `hello` system consists of two tracepoints, `hello_init` and `hello_exit`):

```
$ sudo insmod /lib/modules/$(uname -r)/extra/probes/lttng-probe-hello.ko  
$ lttng list -k | grep hello  
    hello_init (loglevel: TRACE_EMERG (0)) (type: tracepoint)  
    hello_exit (loglevel: TRACE_EMERG (0)) (type: tracepoint)
```

If the custom LTTng module is unloaded (using `sudo rmmod lttng-probe-hello`), the new tracepoints are no longer available. As long as there is a session, running or otherwise, which refers to the kernel events supplied by the custom module, the latter will be locked by the kernel and cannot be unloaded.

Tracepoint availability is completely independent of their occurrence in the kernel or its modules: as stated before, tracepoint instrumentation remains inert until the corresponding tracepoint provider kernel modules are loaded.

The next sections discuss how tracepoints can be added to the kernel (4.6.4), its modules (4.6.5), or a custom module (4.6.6).

4.6.4 Adding LTTng probes to the Linux kernel

4.6.4.1 Building and installing a kernel

The first step in modifying the kernel is, obviously, to obtain its source [32]. If, for instance, a 3.2.0 kernel is installed, choose the `linux-source-3.2.0` package in Synaptic. If some version other than the latest available one is desired, force the package version (Package: Force Version...) to the revision of choice, such as `3.2.0-23.36` (precise). It is also possible to browse the Linux Kernel Archives (<https://www.kernel.org/>) to choose the kernel version. From there, either directly download the tarball (`tar.xz`) or use `wget` to download it (e.g. `wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.9.3.tar.xz` from the destination directory). Untarring the archive can be done with the Archive Manager or from the command line (e.g. `tar -xvJf linux-3.9.3.tar.xz`).

Next, the kernel needs to be configured. This is done like so:

```
$ cd /usr/src/linux-3.9.3
$ make menuconfig
```

For `make menuconfig` to work, the `libncurses` and `libncurses-devel` packages must be already installed. Use `make menuconfig` to browse the kernel options and change them as seems fit (see Figure 36, below). Make sure that the few kernel options LTTng requires are selected (see Section 3.3.3.1). Some defaults are copied from your current kernel's configuration, found in `/boot/config-`uname -r)``, so some warnings may be issued, in particular when the kernel source is for a more recent version than the system's current kernel.

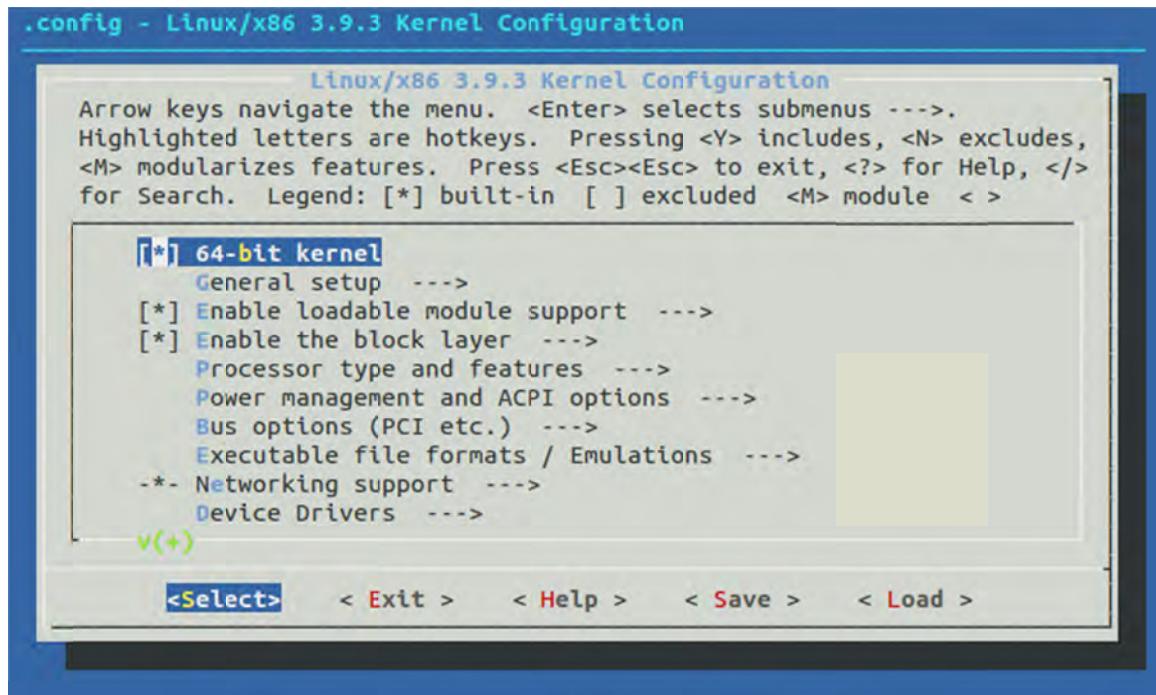


Figure 36: Selecting the kernel options with `make menuconfig`.

Conclude by saving the configuration as a `.config` file (the default name) in the kernel source directory. The build command cannot succeed without a `.config` file. Launch the kernel build with these simple commands:

```
$ make &> make.log  
$ make modules &> make-modules.log
```

The initial build may take a considerable time, while the module build is usually pretty quick. Examine each log to make sure no errors occurred. Install the newly-built kernel alongside the current kernel with these commands:

```
$ sudo make modules_install &> install-modules.log  
$ sudo make install &> install.log
```

The modules should now reside in `/lib/modules/3.9.3` (in the `linux-3.9.3` case) and the new kernel will add the following files to the `/boot` directory:

```
vmlinuz-3.9.3  
System.map-3.9.3  
config-3.9.3  
initrd.img-3.9.3
```

The `vmlinuz` file is the kernel executable compressed image, `System.map` lists the symbols the kernel exports (a subset of the dynamic list one can get from `/proc/kallsyms`; see `--probe` in B.4.10), `config` lists the kernel's configuration options (it is a copy of the `.config` used to build the kernel), and `initrd` is a temporary root file system used during the kernel's boot process. The `make install` command also updates `grub.cfg` to designate the new kernel as the default (as hinted at in the log), so no manual edit is required. When re-building and re-installing a kernel, the previous version has its name suffixed with `.old` (thus `vmlinuz-3.9.3.old`, etc.).

Once the system has rebooted into the new kernel, the `lttng-modules` package must be re-installed to rebuild the LTTng kernel modules against the new kernel. Until this is done, LTTng will report the kernel tracer as “unavailable”, meaning only user-space tracing remains possible. As soon as the `lttng-modules` re-installation is done, kernel tracing becomes available again (the package restarts the `lttng-sessiond` daemon).

As stated in Section 4.6.2, the `drdc lttng-modules` package (if invoked from Synaptic) will do a clean re-install, so any modified or new LTTng tracepoint provider kernel modules would be lost. The kernel would be instrumented, but LTTng would not supply any tracepoint providers to handle the tracepoints. It is thus essential to re-build and re-install manually the `lttng-modules` package (see Section 4.6.2).

If merely modifying your system's current kernel *without changing its name*, LTTng will emerge fully functional after the reboot, because its modules will have remained installed in the still-current path. This can keep the development cycle fairly short: edit the desired kernel source files, design the new tracepoint headers and LTTng kernel tracepoint provider modules (or modify existing ones), run the process-module script (see Topical Note A.14; the script re-builds and re-installs the LTTng modules against the modified kernel source), build and install the updated kernel, reboot.

4.6.4.2 Instrumenting it

Suppose, for example, that the kernel's `sched_switch` event is to be shadowed by a custom `hello_init` event. Assuming the kernel tracepoint header `hello.h` has been written, this is accomplished by editing the `prepare_task_switch` function of `kernel/sched/core.c` (additions in red):

```
[...]
#include "../smpboot.h"

#define CREATE_TRACE_POINTS
#include <trace/events/sched.h>
#include <trace/events/hello.h>
[...]
static inline void
prepare_task_switch(struct rq *rq,
                    struct task_struct *prev,
                    struct task_struct *next)
{
    trace_sched_switch(prev, next);
    trace_hello_init("sched_switch!");
    sched_info_switch(prev, next);
    perf_event_task_sched_out(prev, next);
    fire_sched_out_preempt_notifiers(prev, next);
    prepare_lock_switch(rq, next);
    prepare_arch_switch(next);
}
```

The modified kernel can then be re-built and re-installed.

4.6.5 Adding LTTng probes to a Linux kernel module

This time, the example supposes that the `kvm_async_pf_ready` event is to be shadowed by a custom `hello_init` event (the `kvm.ko` kernel module is the only one currently instrumented in the Linux kernel). Using the same kernel tracepoint header `hello.h`, this is accomplished by editing the `kvm_arch_async_page_present` function of `linux-source/arch/x86/kvm/x86.c` (additions in red):

```
[...]
#include <trace/events/kvm.h>

#define CREATE_TRACE_POINTS
#include <trace/events/hello.h>
#include "trace.h"
[...]
void
kvm_arch_async_page_present(struct kvm_vcpu *vcpu,
                           struct kvm_async_pf *work)
{
    struct x86_exception fault;

    trace_kvm_async_pf_ready(work->arch.token, work->gva);
    trace_hello_init("kvm_async_pf_ready!");
    if (is_error_page(work->page))
        work->arch.token = ~0; /* broadcast wakeup */
    else
        kvm_del_async_pf_gfn(vcpu, work->arch.gfn);
[...]
    vcpu->arch.apf.halted = false;
    vcpu->arch.mp_state = KVM_MP_STATE_RUNNABLE;
}
```

Comparing this example with the previous section's shows the instrumentation proceeds nearly identically as far as editing the code goes. The only difference here is that the `hello.h` include had to be put before the `trace.h` include because of undesirable side-effects of that particular header. The modified kernel can then be re-built and its modules re-installed. That is to say, the 4.6.4.1 procedure can stop with the line:

```
$ sudo make modules_install &> install-modules.log
```

There is no need to do the final `sudo make install`. Instead, it is sufficient to manually unload the newly instrumented module and then reload it. This removes its old image from memory and loads the new image in its place. Depending on the module that was newly instrumented, other dependent modules may need to be unloaded as well. In the example above, it turns out `kvm.ko` is loaded by LTTng, of all things, so one must stop the `lttng-sessiond` service in order to unload `kvm.ko`. The LTTng service can then be restarted (this has the side-effect of reloading `kvm.ko`) and the new events can be traced from that point on. Other, more critical kernel modules may require a reboot to complete the instrumentation process.

4.6.6 Adding LTTng probes to a custom kernel module

This last example will show in detail how to build and instrument a simple kernel module [33]. It will serve to recapitulate the entire process. Here is the `why_hello.c` source code:

```
#include <linux/module.h> // included for all kernel modules
#include <linux/kernel.h> // included for KERN_INFO
#include <linux/init.h> // included for __init and __exit macros

//Module meta-data
MODULE_LICENSE("GPL");
MODULE_AUTHOR("<Your name here>");
MODULE_DESCRIPTION("A simple module");

//Module initialization
static int __init why_hello_init(void)
{
    printk(KERN_INFO "Why, hello!\n");
    return 0;
    // Any other return value means the module couldn't be loaded.
}

//Module finalization
static void __exit why_hello_cleanup(void)
{
    printk(KERN_INFO "Cleaning up why_hello module.\n");
}

//Identify the initialization and finalization handlers
module_init(why_hello_init);
module_exit(why_hello_cleanup);
```

The `printk` function is peculiar to the kernel; its messages go to the kernel log message buffer, which can be read by the `sudo dmesg` shell command. As stated earlier (see Section 4.6.1.3.5), it is similar to `printf` with a few minor differences, the main one being lack of floating-point support. The `KERN_*` macros are simple strings that specify the priority level of the message, ranging from `KERN_EMERG (<0>)` to `KERN_DEBUG (<7>)` (their semantics are given by ‘`syslog(3) level`’; see also the `TRACE_*` logging levels in Section 4.1.2.4). Note that `dmesg` strips the leading logging priority indicator of each message and prefixes the output with a bracketed timestamp instead. The `dmesg` output can be configured and filtered; see the ‘`man dmesg`’ pages for details.

The `why_hello.c` module merely logs its initialization and finalization. Note in passing that a module without a finalization handler won’t be unloaded by the kernel, so you must supply one, even if it is empty. Another key constraint of kernel module programming is that the module can only link to the kernel (the only functions that can be called are the ones exported by the kernel): there are no libraries to link to, unlike with user-space applications.

Here is the `Makefile` of the module:

```
obj-m += why_hello.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Make sure the `make` lines are indented by a tab character (otherwise `make` won't recognise them). Calling `make` (default target `all`) from the working directory where `why_hello.c` lies will compile the kernel module object file `why_hello.ko`. The `Makefile` need not specify compilation instructions for `why_hello.o` since the default is to compile `why_hello.c`. Here is the expected console output of the kernel module compilation:

```
$ cd /home/user/module
$ make
make -C /lib/modules/3.2.0-53-generic/build M=/home/user/module modules
make[1]: Entering directory "/usr/src/linux-headers-3.2.0-53-generic"
  CC [M]  /home/user/module/why_hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/user/module/why_hello.mod.o
  LD [M]  /home/user/module/why_hello.ko
make[1]: Leaving directory "/usr/src/linux-headers-3.2.0-53-generic"
```

The console session shown below uses `dmesg` to clear the kernel message buffer, loads the module, displays the kernel message buffer, unloads the module and displays the kernel message buffer again:

```
$ sudo dmesg --clear
$ sudo insmod why_hello.ko
$ sudo dmesg
[74774.087796] Why, hello!
$ sudo rmmod why_hello
$ sudo dmesg
[74774.087796] Why, hello!
[74790.810844] Cleaning up why_hello module.
```

The `insmod` command inserts the module in the kernel, while the `rmmod` command removes it. The unprivileged `lsmod` command can be used to list the kernel's currently loaded modules (this list can be fairly large, so piping it to `grep` is usually a good idea). The `mod-probe` command can do all that these three `*mod` commands can do, and then some—it will for instance load modules and their dependencies in intelligent order, unlike `insmod`. The unprivileged `modinfo` command, finally, displays information about the module, such as its author, description, dependencies, etc.

If the module was compiled incorrectly (against a different kernel version or configuration, for instance), insmod will issue an error message (insmod will leave a more detailed message in the dmesg buffer):

```
insmod: error inserting '<module>.ko': -1 Invalid module format
```

Instrumenting the module follows the same simple pattern as before (added lines in red):

```
#include <linux/module.h> // included for all kernel modules
#include <linux/kernel.h> // included for KERN_INFO
#include <linux/init.h> // included for __init and __exit macros
#define CREATE_TRACE_POINTS
#include <trace/events/hello.h>

//Module meta-data
MODULE_LICENSE("GPL");
MODULE_AUTHOR("<Your name here>");
MODULE_DESCRIPTION("A simple module");

//Module initialization
static int __init why_hello_init(void)
{
    trace_hello_init("Why, hello!");
    printk(KERN_INFO "Why, hello!\n");
    return 0;
    // Any other return value means the module couldn't be loaded.
}

//Module finalization
static void __exit why_hello_cleanup(void)
{
    trace_hello_exit("Bye!");
    printk(KERN_INFO "Cleaning up why_hello module.\n");
}

//Identify the initialization and finalization handlers
module_init(why_hello_init);
module_exit(why_hello_cleanup);
```

The make of the instrumented why_hello.c will yield the same log as the uninstrumented version.

Here is the tracepoint header (`hello.h`) that was used:

```
//The LTTng tracepoint provider kernel module will be
//named 'lttng-probe-TRACE_SYSTEM'
#undef TRACE_SYSTEM
#define TRACE_SYSTEM hello

#if !defined(_TRACE_HELLO_H) || defined(TRACE_HEADER_MULTI_READ)
#define _TRACE_HELLO_H

#include <linux/tracepoint.h>

DECLARE_EVENT_CLASS(hello_template,
    TP_PROTO(const char *astring),
    TP_ARGS(astring),
    TP_STRUCT_entry(
        __string(the_string, astring)
    ),
    TP_fast_assign(
#ifdef _LTTNG_MAINLINE
        __assign_str(the_string, astring);
#else
        tp strcpy(the_string, astring)
#endif
    ),
    TP_printk("%s", __get_str(the_string))
) _LTTNG_SEPARATOR

DEFINE_EVENT(hello_template, hello_init,
    TP_PROTO(const char *astring), TP_ARGS(astring)
) _LTTNG_SEPARATOR

DEFINE_EVENT(hello_template, hello_exit,
    TP_PROTO(const char * astring), TP_ARGS(astring)
) _LTTNG_SEPARATOR

#endif /* _TRACE_HELLO_H */

/* This part must be outside protection */
#undef _LTTNG_SEPARATOR
#ifdef _LTTNG_MAINLINE
#include <trace/define_trace.h>
#else
#include "../../probes/define_trace.h"
#endif
```

Note the `#ifdef _LTTNG_MAINLINE` which serves to translate the `TP_fast_assign` for the `lttng-module` destination. Note also that the file above is incomplete by itself. It was used in combination with the `process-module` script of Topical Note A.14 which takes care of copying the stub to its two required locations, `linux-headers` and `lttng-module`, while prefixing it with the appropriate lines:

```
//These two #defines handle the subtle differences between
//the lttng-module and mainline versions of this file.
//For mainline:
#define _LTTNG_MAINLINE
#define _LTTNG_SEPARATOR ;
```

or

```
//These two #defines handle the subtle differences between
//the lttng-module and mainline versions of this file.
//For lttng-module:
#undef _LTTNG_MAINLINE
#define _LTTNG_SEPARATOR /* */
```

Here is the tracepoint handler module `lttng-probe-hello.c` (as per Section 4.6.2):

```
#include <linux/module.h>

/*
 * Create the tracepoint static inlines from the kernel to validate
 * that the trace event macros match the kernel we run on.
 */
#include <trace/events/hello.h>

#include "../wrapper/tracepoint.h"

/*
 * Create LTTng tracepoint probes.
 */
#define LTTNG_PACKAGE_BUILD
#define CREATE_TRACE_POINTS
#define TRACE_INCLUDE_PATH ../instrumentation/events/lttng-module

#include "../instrumentation/events/lttng-module/hello.h"

MODULE_LICENSE("GPL and additional rights");
MODULE_AUTHOR("Daniel U. Thibault <daniel.thibault@drdc-rddc.gc.ca>");
MODULE_DESCRIPTION("LTTng hello probes");
```

The module is added to `lttng-modules/probes/Makefile` (as per Section 4.6.2):

```
[...]
obj-m += lttng-probe-statedump.o
obj-m += lttng-probe-hello.o
[...]
```

In the following example, the local folder where the module (`why_hello.c`), tracepoint header (`hello.h`) and tracepoint handler module (`lttng-probe-hello.c`) reside is `/home/user/Documents/mymodule` (copies are installed by the `lttng-ust drdc` package in the `/usr/src/lttng-ust/doc/examples/drdc/kernel` directory). The `lttng-modules` folder is `/usr/src/lttng-modules` and the Linux headers were installed in `/usr/src/linux-headers`. The compilation, installation and testing session follows:

```
$ cd /home/user/Documents/mymodule
$ LINUXHEADERS=/usr/src/linux-headers
$ export TRACE_SYSTEM=hello
$ export TARGET=$LINUXHEADERS/include/trace/events/$TRACE_SYSTEM.h
$ gksudo -E -- sh -c 'cat > "$TARGET" << ENDENDEND
//These two #defines handle the subtle differences between
//the lttng-module and mainline versions of this file.
//For mainline:
#define _LTTNG_MAINLINE
#define _LTTNG_SEPARATOR ;

`cat "./$TRACE_SYSTEM.h"
ENDENDEND'
$ LTTNGMODULE=/usr/src/lttng-modules/instrumentation/events/lttng-module
$ TARGET=$LTTNGMODULE/$TRACE_SYSTEM.h
sudo -E -- sh -c 'cat > "$TARGET" << ENDENDEND
//These two #defines handle the subtle differences between
//the lttng-module and mainline versions of this file.
//For lttng-module:
#undef _LTTNG_MAINLINE
#define _LTTNG_SEPARATOR /* */

`cat "./$TRACE_SYSTEM.h"
ENDENDEND'
$ sudo cp lttng-probe-hello.c \
  /usr/src/lttng-modules/probes/lttng-probe-hello.c
$ make &> make.log
$ sudo dmesg --clear
```

```

$ cd /usr/src/lttng-modules
$ sudo make &> make-hello.log
$ sudo make modules_install &> install-hello.log
$ lttng list -k | grep hello
$ sudo insmod /lib/modules/`uname -r`/extra/probes/lttng-probe-hello.ko
$ lttng list -k | grep hello
    hello_init (loglevel: TRACE_EMERG (0)) (type: tracepoint)
    hello_exit (loglevel: TRACE_EMERG (0)) (type: tracepoint)
$ lttng create demo
Session demo created.
Traces will be written in /home/user/lttng-traces/demo-20140212-085209
$ lttng enable-event -k hello_init
kernel event hello_init created in channel channel0
$ lttng enable-event -k hello_exit
kernel event hello_exit created in channel channel0
$ lttng start
Tracing started for session demo

$ cd /home/user/Documents/mymodule
$ sudo insmod ./why_hello.ko
$ sudo dmesg
[581761.454738] Why, hello!
$ sudo rmmod why_hello
$ sudo dmesg
[581761.454738] Why, hello!
[581771.769545] Cleaning up why_hello module.

$ lttng destroy
Session demo destroyed
$ sudo rmmod lttng-probe-hello
$ babeltrace /home/user/lttng-traces/demo-20140212-085209 \
-w /home/user/lttng-traces/demo_na_fa --names all --fields all

```

The unavoidably convoluted `gksudo sh` command copies the tracepoint header to the appropriate `linux-headers` subdirectory while prefixing it with the crucial missing defines. Another copy should ideally be made in `/usr/src/lttng-modules/instrumentation/events/mainline`, but it is not strictly necessary. The second `sudo sh` command does the same for the `lttng-modules` subdirectory, using the appropriate defines. The conventional `sudo cp` command copies the tracepoint provider kernel module source code to the `probes` subdirectory of `lttng-modules`. The `probes/Makefile` was previously modified to add `lttng-probe-hello.c` to the list of modules to compile, as stated earlier (see also Section 4.6.2). The first `make` command builds the `hello.ko` module (not shown is the inspection of the log of the build, `make.log`, which confirms that the build was successful). The first `dmesg` command clears the kernel log buffer.

The working directory is then switched to `lttng-modules` and the next two `make` commands rebuild and re-install the LTTng kernel modules. Inspection of those logs (`make-hello.log` and `install-hello.log`) is not shown. The first log is very short, since only the new `lttng-probe-hello` module needed to be compiled. The list of available kernel tracepoints that include `hello` in part of their names is shown before and after loading the `lttng-probe-hello` module, confirming that the new events `hello_init` and `hello_exit` have become available. A `demo` tracing session is then created to track just those two events and started.

The working directory is then switched back to `mymodule` and the `why_hello.ko` module loaded. The kernel log buffer confirms that module initialization occurred. The `why_hello` module is then unloaded, the kernel log buffer confirming that module finalization occurred.

The tracing session is concluded and the `lttng-probe-hello` module unloaded. Finally, the trace is converted using `babeltrace`. Here are the contents of the resulting `demo_na_fa`:

```
timestamp = 08:52:32.078461761, delta = +?.?????????,  
trace = /home/user/lttng-traces/demo-20140212-085209/kernel,  
trace:hostname = hostname, trace:domain = kernel,  
name = hello_init, stream.packet.context = { cpu_id = 0 },  
event.fields = { message = "Hello!" }  
timestamp = 08:52:42.393270336, delta = +10.314808575,  
trace = /home/user/lttng-traces/demo-20140212-085209/kernel,  
trace:hostname = hostname, trace:domain = kernel,  
name = hello_exit, stream.packet.context = { cpu_id = 0 },  
event.fields = { message = "Bye!" }
```

The Topical Note A.14 shell script `process-lttng-system` (also found in the `/usr/src/lttng-ust/doc/examples/drdc/kernel` directory) automates the entire process.

This page intentionally left blank.

References

- [1] IEEE 610.12-1990 (revising 729-1983 [misprinted as 792-1983]), *IEEE Standard Glossary of Software Engineering Terminology*, 1990
- [2] IEEE 100-2000, *The Authoritative Dictionary of IEEE Standards Terms, Seventh Edition*, IEEE, 2000
- [3] ISO/IEC/IEEE 24765:2010(E), *Systems and software engineering — Vocabulary (Ingénierie des systèmes et du logiciel — Vocabulaire)*, ISO/IEC/IEEE, 2010
- [4] Mathieu Desnoyers, Julien Desfossez, and David Goulet, *LTTng 2.0: Tracing for power users and developers – Part 1*, LWN.net, April 11, 2012, <http://lwn.net/Articles/491510/>
- [5] Mathieu Desnoyers, *LTTng 2.0 Low-Overhead Tracing Architecture* diagram, 2011
http://lttng.org/sites/lttng.org/files/LTTng2_0Architecture_pa3.pdf
- [6] Mathieu Desnoyers, *Re: [lttng-dev] Making 32-bit user-space events on a 64-bit Linux system*, lttng-dev Digest, vol. 66, no. 7, 08 oct 2013 <http://www.mail-archive.com/lttng-dev@lists.lttng.org/msg04550.html>
- [7] Mathieu Desnoyers, *Low-Impact Operating System Tracing*, Ph. D. dissertation, Dépt. de génie informatique et génie logiciel, École Polytechnique de Montréal, 2009
- [8] David Goulet, *Re: [lttng-dev] lttng-dev Digest, Vol 67, Issue 48*, lttng-dev Digest, vol. 70, no. 9, 4 feb 2014 <http://www.mail-archive.com/lttng-dev@lists.lttng.org/msg05286.html>
- [9] Daniel U. Thibault, *Exclusion of '/../' should occur earlier during trace creation*, LTTng bug #721, 2014-Jan-17, <http://bugs.lttng.org/issues/721>
- [10] Jim Keniston, Prasanna S. Panchamukhi, Masami Hiramatsu, *Kernel Probes (Kprobes)*, 2013-Aug-12, <https://www.kernel.org/doc/Documentation/kprobes.txt>
- [11] Mathieu Desnoyers, Julien Desfossez, and David Goulet, *LTTng 2.0: Tracing for power users and developers – Part 2*, LWN.net, April 18, 2012, <http://lwn.net/Articles/492296/>
- [12] Sudhanshu Goswami, *An introduction to KProbes*, LWN.net, April 18, 2005, <http://lwn.net/Articles/132196/>
- [13] Masami Hiramatsu, *Kprobe-based Event Tracing*, <http://os1a.cs.columbia.edu/lxr/source/Documentation/trace/kprobetrace.txt> consulted 2013-07-25
- [14] Srikar Dronamraju, *Uprobe-tracer: Uprobe-based Event Tracing*, LWN.net, May 30, 2012, <http://lwn.net/Articles/499286/>

- [15] Jim Keniston, Prasanna S. Panchamukhi, Masami Hiramatsu, *Kernel Probes (Kprobes)*, <https://www.kernel.org/doc/Documentation/kprobes.txt> consulted 2013-07-17
- [16] Daniel P. Bovet, Marco Cesati, *Understanding the Linux Kernel*, O'Reilly, Media Inc., Sebastopol (CA), 2006 (3rd edition)
- [17] Jan Glauber, *[lttng-dev] [Patch lttng-modules v2] ARM: Resolve sys_unknown system calls*, lttng-dev Digest, Vol 72, Issue 38, 14 April 2014
- [18] Mathieu Desnoyers, *Re: What are sys calls starting with compat?*, lttng-dev Digest, vol. 68, no. 21, 10 dec 2013 <http://www.mail-archive.com/lttng-dev@lists.lttng.org/msg05116.html>
- [19] Simon Marchi, *[lttng-dev] Names collisions with LTTng-UST*, lttng-dev digest, vol. 63, no. 58, 31 July 2013 <http://www.mail-archive.com/lttng-dev@lists.lttng.org/msg04022.html>
- [20] Julien Desfossez, *Re: [lttng-dev] lttng snapshots and running traces*, lttng-dev Digest, vol. 65, no. 45, 2013-sept-26 <http://www.mail-archive.com/lttng-dev@lists.lttng.org/msg04503.html>
- [21] Daniel U. Thibault, *Disambiguating fully qualified event names, timestamp-sensitive metadata*, LTTng bug #720, 2014-Jan-16, <http://bugs.lttng.org/issues/720>
- [22] Daniel U. Thibault, *Something is fishy with truncated event names*, LTTng bug #648, 2013-Oct-10, <http://bugs.lttng.org/issues/648>
- [23] JTC (Joint Technical Committee) 1/SC (Sub-Committee) 22/WG (Working Group) 14, *Programming languages — C (C99) WG14/N1124 Committee Draft (2005 May 6) ISO/IEC 9899:TC2 (9899:1999)*, 2005
- [24] Geneviève Bastien, *[lttng-dev] [Patch LTTng-ust] RFC: Syntax changes to add support of other CTF metadata*, lttng-dev Digest, vol 68, no. 11, 2013-dec-06 <http://www.mail-archive.com/lttng-dev@lists.lttng.org/msg05076.html>
- [25] Daniel U. Thibault, *Can't trace long double payload field?*, LTTng bug #473, 2013-Mar-12, <http://bugs.lttng.org/issues/473>
- [26] Oracle, *Java Native Interface Specification*, 1993–2013, <http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html>, accessed 2013-07-31
- [27] Chua Hock-Chuan, *Java Programming Tutorial: Java Native Interface (JNI)*, July 2012, <http://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html>, accessed 2013-07-31
- [28] *Java Native Access (JNA)*, <https://github.com/twall/jna#readme>
- [29] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, *The Java Language Specification, Third Edition*, Sun Microsystems, Inc., Santa Clara (CA), 2005

- [30] Oracle, *Java SE Documentation, Chapter 3: JNI Types and Data Structures, Primitive Types* <http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html#wp428>
- [31] Daniel U. Thibault, *making ltng-tools with-consumerd-only runs into trouble with the tests*, LTTng bug #647, 2013-Oct-03, <http://bugs.lttng.org/issues/647>
- [32] Lakshmanan Ganapathy, *How to Compile Linux Kernel from Source to Build Custom Kernel*, The Geek Stuff, 13 June 2013, <http://www.thegeekstuff.com/2013/06/compile-linux-kernel/>
- [33] Lakshmanan Ganapathy, *How to Write Your Own Linux Kernel Module with a Simple Example*, The Geek Stuff, 17 July 2013, <http://www.thegeekstuff.com/2013/07/write-linux-kernel-module/>
- [34] Luvr (Luc Van Rompaey), Creating a Trusted Local Repository from which Software Updates can be installed, Ubuntu Forums, *Ubuntu Specialised Discussions: Tutorials*, 08 Mar 2009, <http://ubuntuforums.org/showthread.php?t=1090731>
- [35] Michael Kerrisk, *Namespaces in operation, part 1: namespaces overview*, LWN.net, January 4, 2013 <http://lwn.net/Articles/531114/>
- [36] Michael Kerrisk, *Namespaces in operation, part 2: the namespaces API*, LWN.net, January 8, 2013 <http://lwn.net/Articles/531381/>
- [37] Michael Kerrisk, *Namespaces in operation, part 3: PID namespaces*, LWN.net, January 16, 2013 <http://lwn.net/Articles/531419/>
- [38] Michael Kerrisk, *Namespaces in operation, part 4: more on PID namespaces*, LWN.net, January 23, 2013 <http://lwn.net/Articles/532748/>
- [39] Michael Kerrisk, *Namespaces in operation, part 5: User namespaces*, LWN.net, February 27, 2013 <http://lwn.net/Articles/532593/>
- [40] Michael Kerrisk, *Namespaces in operation, part 6: more on user namespaces*, LWN.net, March 6, 2013 <http://lwn.net/Articles/540087/>
- [41] Alex Gonzalez, *Debugging the Linux kernel using ftrace*, Lindus Embedded, April 14, 2010, <http://www.lindusembedded.com/blog/2010/04/14/debugging-the-linux-kernel-using-ftrace/>
- [42] Tim Bird, *Measuring Function Duration on ARM with Ftrace*, Linux Symposium 2009, <http://elinux.org/images/d/d6/Measuring-function-duration-with-ftrace.pdf>
- [43] Mathieu Desnoyers, *[ltng-dev] [PATCH 10/15] ltng: remove ftrace function tracer support (but keep, ltng-dev Digest, vol. 43, no. 74, 2011-nov-30* <http://www.mail-archive.com/ltng-dev@lists.lttng.org/msg00109.html>
- [44] Daniel U. Thibault, *ltng-syscalls-generate-headers.sh is broken*, LTTng bug #771, 2014-Mar-25, <http://bugs.lttng.org/issues/771>

- [45] Daniel U. Thibault, *Synaptic won't import key files*, Ubuntu bug #855552, 21 Sep 2011, <https://bugs.launchpad.net/ubuntu/+source/synaptic/+bug/855552>
- [46] Artem Anisimov, *Build fails if gold is used as linker and some libraries reside in [sic] /usr/local/lib*, GCC Bugzilla Bug 42756, 15 January 2010, http://gcc.gnu.org/bugzilla/show_bug.cgi?id=42756
- [47] lttng-ust: README
- [48] POSIX.1-2008 (IEEE 1003.1-2008), *Standard for Information Technology — Portable Operating System Interface (POSIX®), Base Specifications, Issue 7*, IEEE, 2008, http://cfajohnson.com/pdf/Portable_Operating_System_Interface-POSIX.pdf
- [49] Daniel U. Thibault, *lttng verbosity*, LTTng bug #748, 2014-Mar-07, <http://bugs.lttng.org/issues/748>
- [50] Daniel U. Thibault, *[lttng-dev] Is the lttng --group command option useless?*, lttng-dev Digest, vol. 63, no. 26, 2013-jul-09 <http://www.mail-archive.com/lttng-dev@lists.lttng.org/msg03876.html>
- [51] Daniel U. Thibault, *lttng list <session> should list the contexts added to each channel*, LTTng bug #749, 2014-Mar-07, <http://bugs.lttng.org/issues/749>
- [52] Mathieu Desnoyers, *Personal communication*, 6 May 2013 22:59
- [53] R. Hinden, B. Carpenter, L. Masinter, *Request for Comments: 2732: Format for Literal IPv6 Addresses in URL's [sic]*, Internet Engineering Task Force, December 1999, <http://www.ietf.org/rfc/rfc2732.txt>
- [54] Wikipedia, *IPv6 address: Presentation*, http://en.wikipedia.org/wiki/IPv6_address#Presentation 2013 December 5
- [55] Daniel U. Thibault, *disable-event -u -a on a channel-less session creates the domain but not the default channel*, LTTng bug #638, 2013-Sep-20, <http://bugs.lttng.org/issues/638>
- [56] Daniel U. Thibault, *Channel names ought to be filtered just like session names*, LTTng bug #751, 2014-Mar-07, <http://bugs.lttng.org/issues/751>
- [57] Daniel U. Thibault, *utils_parse_size_suffix suffers from several problems*, LTTng bug #633, 2013-Sep-18, <http://bugs.lttng.org/issues/633>
- [58] Daniel U. Thibault, *lttng create --subbuf-size, --tracefile-size and --num-subbuf have a rounding problem*, LTTng bug #641, 2013-Sep-24, <http://bugs.lttng.org/issues/641>
- [59] Daniel U. Thibault, *LTTng memory allocation failure goes unreported*, LTTng bug #653, 2013-Oct-21, <http://bugs.lttng.org/issues/653>

- [60] Daniel U. Thibault, *ltng list <session> should give the tracefile-size and tracefile-count of the channels*, LTTng feature #642, 2013-Sep-24, <http://bugs.lttng.org/issues/642>
- [61] Mathieu Desnoyers, *Re: [ltng-dev] ltng, ltng-dev Digest*, vol. 61, no. 11, 2013-May-03 <http://www.mail-archive.com/ltng-dev@lists.lttng.org/msg03276.html>
- [62] Daniel U. Thibault, *Re-enabling an event is treated inconsistently*, LTTng bug #639, 2013-Sep-24, <http://bugs.lttng.org/issues/639>
- [63] Daniel U. Thibault, *Kernel event name shadowing can break syscall tracing*, LTTng bug #658, 2013-Nov-05, <http://bugs.lttng.org/issues/658>
- [64] Daniel U. Thibault, *'ltng list session_name' should list the filters attached to the events*, LTTng feature #630, 2013-Sep-11, <http://bugs.lttng.org/issues/630>
- [65] Daniel U. Thibault, *Error messages are issued when enabling all events in a new channel*, LTTng bug #644, 2013-Sep-25, <http://bugs.lttng.org/issues/644>
- [66] Salman Rafiq, *Re: [ltng-dev] Reading CTF trace using Babeltrace API*, ltng-dev Digest, vol. 66, no. 19, 2013-Oct-23 <http://www.mail-archive.com/ltng-dev@lists.lttng.org/msg04602.html>
- [67] Daniel U. Thibault, *In the kernel domain, the loglevel command option is treated inconsistently*, LTTng bug #645, 2013-Sep-25, <http://bugs.lttng.org/issues/645>
- [68] Daniel U. Thibault, *loglevel and loglevel-only filtering are indistinguishable from the console*, LTTng bug #628, 2013-Sep-06, <http://bugs.lttng.org/issues/628>
- [69] Daniel U. Thibault, *Inconsistent filter bytecode rules*, LTTng bug #688, 2013-Nov-19, <http://bugs.lttng.org/issues/688>
- [70] Jérémie Galarneau, *Re: [ltng-dev] Enable UST probes only in specific process*, ltng-dev Digest, Vol 75, Issue 40, 31 July 2014
- [71] Daniel U. Thibault, *'ltng list -u session_name' doesn't behave as expected*, LTTng bug #654, 2013-Oct-21, <http://bugs.lttng.org/issues/654>
- [72] Julien Desfossez, *Re: [ltng-dev] ltng snapshots and running traces*, ltng-dev Digest, vol. 65, no. 42, 2013-Sep-26 <http://www.mail-archive.com/ltng-dev@lists.lttng.org/msg04495.html>
- [73] Daniel U. Thibault, *ltng snapshot record fails to increment the snapshot sequence number*, LTTng bug #660, 2013-Nov-15, <http://bugs.lttng.org/issues/660>
- [74] Julien Desfossez, ltng-dev Digest, vol. 65, nos. 42 through 46, 2013-Sep-26

- [75] Daniel U. Thibault, *snapshot add-output does not report the default name of the output set*, LTTng bug #634, 2013-Sep-18, <http://bugs.lttng.org/issues/634>
- [76] Daniel U. Thibault, *lttng snapshot record can erroneously claim a successful snapshot*, LTTng bug #606, 2013-Jul-29, <http://bugs.lttng.org/issues/606>
- [77] Daniel U. Thibault, *lttng snapshot record fails if no "output" is set*, LTTng bug #608, 2013-Jul-29, <http://bugs.lttng.org/issues/608>
- [78] Daniel U. Thibault, *lttng snapshot record fails for --buffers-pid traces*, LTTng bug #607, 2013-Jul-29, <http://bugs.lttng.org/issues/607>
- [79] Daniel U. Thibault, *lttng snapshot record ignores user-space if kernel is also being traced*, LTTng bug #661, 2013-Nov-15, <http://bugs.lttng.org/issues/661>
- [80] Daniel U. Thibault, *lttng snapshot record can erroneously claim a successful snapshot (again!)*, LTTng bug #662, 2013-Nov-15, <http://bugs.lttng.org/issues/662>
- [81] Mathieu Desnoyers, *Personal communication*, 7 May 2013 11:42
- [82] Daniel U. Thibault, *babeltrace core dumps when using -i dummy*, LTTng bug #462, 2013-Feb-23, <http://bugs.lttng.org/issues/462>
- [83] Daniel U. Thibault, *babeltrace output is more irregular than required*, LTTng bug #532, 2013-May-15, <http://bugs.lttng.org/issues/532>
- [84] Sheng Yang , *Extending KVM with new Intel® Virtualization technology*, 12 June 2008, KVM Forum 2008, Napa Valley, CA [http://www.linux-kvm.org/wiki/images/c/c7/KvmForum2008\\$kdf2008_11.pdf](http://www.linux-kvm.org/wiki/images/c/c7/KvmForum2008$kdf2008_11.pdf)
- [85] IEEE 729-1983, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1983 (superseded by IEEE 610.12-1990)

Annex A Configurations and notes

This annex regroups topical notes, covering topics which are only tangentially relevant to LTTng and tracing in general. Some are little more than cheat sheets summarising relevant commands (e.g. A.4), some deal with circumventing known bugs affecting some configurations (e.g. A.11) or technical restrictions specific to an organisation’s networking policies (e.g. A.15), some speculate about the forthcoming evolution of Linux (e.g. A.5), etc.

A.1 LTTng bash command completion

Beside the creation of `/etc/init/lttng-sessiond.conf` and of the tracing group, there is one more thing which an installation of LTTng from `git.lttng.org` (Section 3.3) will miss. The `lttng-tools` package installs `/etc/bash_completion.d/lttng`, which makes possible auto-completion of `lttng` command lines. Since this is quite handy, you may want to fetch the package just in order to extract that file from it.

To do so, select the package for installation within Synaptic, then click `Apply`. In the `Summary` dialog, check the ‘Download package files only’ box and then click the dialog’s `Apply` button. This downloads the package and any other packages it depends on to Synaptic’s cache but does not install them. The cache is located at `/var/cache/apt/archives`. Open the `.deb` package file with the Archive Manager and extract the files you need to a staging folder (e.g. `/home/<username>/Downloads`). The GUI’s contextual menu offers an “Extract Here” entry which will deploy the package’s entire contents as a subfolder of the package’s. You can now move the files you need to their ultimate destinations. Several of these may require super-user privileges, so you will need to either open a super-user Nautilus window (`gksudo xdg-open /home/<username>/Downloads`) or use the command line (`sudo cp /home/<username>/Downloads/<file> /<destination_path>`). Substitute the appropriate values for `<username>`, `<file>`, and `<destination_path>`.

A.2 Making your own installation packages

Several tools exist to help you make Debian installation packages (or other standard schemes such as RPM and Slackware): `checkinstall`, `dpkg-buildpackage`, `pbuilder`, etc. This topic is beyond the scope of this endnote, however. The DVD-ROM’s software repository packages were made using the existing Ubuntu `lttng` packages as templates, substituting the appropriate source files and hand-crafting the installation scripts in order to remain as universally applicable as possible.

The package templates were deployed (`deb` files are merely standard Unix `ar` archives with a specific internal structure) and later repackaged with the ‘`dpkg --build <packagename>`’ command.

The resulting .deb files can be installed directly by the Ubuntu Software Centre or by the sudo dpkg -i <package>.deb command line. To make the packages visible in Synaptic, you must create a local repository and register it with Synaptic, a simple matter described in Section A.2.2.

A.2.1 Making a repository

The DVD-ROM accompanying this document contains a repository of the LTTng packages of choice, 2.3.0 (2013-Sep-06), prepared as follows. From the root drdc folder, a folder was created for each git source (git.lttng.org and git.efficios.com). In each, a ubuntu/pool/main path was created with sub-folders for each set of deb packages by initial (e.g. l/lttng-modules-2.3.0-1.deb, u/userspace-rcu-0.8.0-1.deb and so on). The ubuntu/dists/precise/main/binary-all and ...main/source paths were also created. This structure mimics that used by the Ubuntu repositories. The following commands (requiring the dpkg-dev package) were then run from each ubuntu folder:

```
$ dpkg-scanpackages pool /dev/null > \
    dists/precise/main/binary-all/Packages
$ gzip -9c dists/precise/main/binary-all/Packages > \
    dists/precise/main/binary-all/Packages.gz
$ dpkg-scansources pool /dev/null > \
    dists/precise/main/source/Sources
$ cp dists/precise/main/binary-all dists/precise/main/binary-amd64
$ cp dists/precise/main/binary-all dists/precise/main/binary-i386
$ ../../../../drdc-make-release
```

The last command listed above, drdc-make-release, is a special-purpose script which implements the procedure outlined in [34]. Briefly, a Release file is constructed, containing in particular the md5sums of the Packages and their sizes (obtained using wc --bytes). This file is then signed using gpg --armor --detach-sign --output Release.gpg --local-user <keyname> Release (gpg's key ring is distinct from apt's and must be loaded beforehand with the public *and* secret keys: gpg --import <secretkeyfile>.asc). The <keyname> is either the uid listed by gpg --list-secret-keys (e.g. "First M. Last <email@provider>") or the hash part of the pub listed by the same command, prefixed by 0x (e.g. 0x059B302F). This important bit is not documented in gpg's man pages.

By importing the key used (ID 059B302F; see A.8.1 for the procedure to import the key into Synaptic), you can make the drdc repository trusted. This is not strictly necessary and serves only to skip the “Not Authenticated” warnings you will otherwise get.

The repository is best copied to a convenient folder before being added to Synaptic's list of repositories.

A.2.2 Using the repository

You can add the repository to the Synaptic list by invoking Settings: Repositories: Other Software: Add... (from the Ubuntu Software Centre, choose Edit: Software Sources...: Other Software: Add...) and specify the following APT lines (one at a time):

```
$ deb file:///<repositories>/git.lttng.org/ubuntu precise main  
$ deb file:///<repositories>/git.efficios.com/ubuntu precise main
```

Where <repositories> is the path to the copy of drdc you made on your file system. Optionally, uncheck the “(Source Code)” lines which Synaptic adds as a result of this action (the DVD-ROM repository does not contain source code in the expected sense). After Synaptic is Reloaded, the packages from the DVD-ROM will appear in the list. If working from the command line, you will need to edit /etc/apt/sources.list and append the APT lines at its end (in the “third-party developers” section; editing sources.list requires super-user privileges). The Ubuntu GUI opens /etc/apt/sources.list directly with the Software Sources gadget. Note that adding a source for `main` is done like with “Other Software” but ends up setting the “Download from” drop-down of the “Ubuntu Software” tab.

The Ubuntu packages bearing the same names are hidden by the new versions supplied by the repository (i.e. all but `lttng-modules`, since the Ubuntu package’s name is different: `lttng-modules-dkms`). Note that a Web repository is prepared very similarly, except that packages must be gzipped.

To make the repository trusted, follow the procedure outlined in Endnote A.8.1. This is not strictly necessary and serves only to skip the “Not Authenticated” warnings you will otherwise get.

A.3 Trace collisions

The current LTTng trace naming scheme (see Section 2.6.2) is susceptible to occasional collisions. Recall that the three possible naming schemes within an `lttng-traces` directory are:

```
[<host>/]<session>-<time>/ [<snapshot>-<time>-<seq>/]  
    kernel/  
  
[<host>/]<session>-<time>/ [<snapshot>-<time>-<seq>/]  
    ust/uid/<userID>/<bitness>/  
  
[<host>/]<session>-<time>/ [<snapshot>-<time>-<seq>/]  
    ust/pid/<process>-<processID>-<time>/
```

It should first be noted that the `output` command option of the `create` command can be used to override the `<session>-<time>` part (as well as the `lttng-traces` part). This means two session daemons can readily “force” a collision. The sessions need not be contemporaneous of each other, which also means they could both be created by a single daemon.

The first potential collision is with <host>. A remote session issued by *host* and named *session* will collide with a local session named *host/session* if both are created within the same one-second window. This is easily avoided by forgoing use of the folder separator within session names.

The second potential collision is with <session>-<time>. If a session is created and destroyed within a one-second window, creating another session of the same name will result in a collision if also done within the same one-second window. This is essentially impossible with a human operator but could occur if LTTng were driven by a software system. This can also occur if two session daemons create homonymous sessions within the same one-second window in the same `lttng-traces` directory (this normally leads to `.lttngrc` contention as well).

No collisions can occur between same-session snapshots thanks to the <seq> part of the name, even when the timestamps are the same.

The third potential collision is with <process>-<processID>-<time>, and can only occur if a process dynamically loads, unloads and reloads a tracepoint provider within a one-second window. The <processID> part of the name ensures no two processes can collide even though they have the same <process> part (which is rather a common occurrence). The process must unload its tracepoint provider using `dlclose()` to terminate the first instance of `pid/<process>-<processID>-<time>/`, which is only possible if it was loaded using `dlopen()` in the first place. If the process loads its first tracepoint provider, unloads it and loads a new one (or the same one) within a one-second window, the second instance of `pid/<process>-<processID>-<time>` will collide with the first.

In summary, trace collisions are possible but unlikely, unless mischief (or very poor planning) is at work.

A.4 User groups

User groups are a convenient way of organising user accounts for the management of their security privileges.

A.4.1 Group membership

You can find out which groups a user belongs to with this command:

```
$ groups [<username>]
```

The <username> can be omitted: it defaults to the invoking user. To simply check for tracing group membership, use this command (which returns nothing if the <username> does not belong to it):

```
$ groups [<username>] | grep tracing
```

A.4.2 Listing groups

You can list all groups alphabetically with this command:

```
$ getent group | cut -d: -f1,3 | sort
```

To list the groups numerically:

```
$ getent group | cut -d: -f1,3 | sort -g -t: -k2
```

To simply check for the tracing group's existence, use this command (which returns nothing if the group does not exist):

```
$ getent group | grep tracing
```

A.4.3 Group creation

To create the group, you can use `groupadd` or `addgroup` (the former is an application, the latter a Perl script; the command lines coincide in this case):

```
$ sudo groupadd --system tracing  
$ sudo addgroup --system tracing
```

A.4.4 Joining and quitting a group

In previous versions of Ubuntu with classic Gnome desktop, one could easily add users to groups (and remove them from groups) with the user management tool. Not anymore. With the Unity desktop, that tool is gone and currently the only way to manage groups in Ubuntu is through the command-line.

To add a user to the tracing group, you use `usermod` or `adduser` (the former is an application, the latter a Perl script; neither command can handle more than one `<username>` at a time):

```
$ sudo usermod -a -G tracing <username>  
$ sudo adduser <username> tracing
```

Conversely, a user is removed from a group with:

```
$ sudo deluser <username> <group_name>
```

Users who are logged in when joined to (or removed from) the tracing group must log out and back in to update their group membership.

A.5 Namespaces

Namespaces, progressively introduced in the Linux kernel starting with 2.6.23 (circa 2008) and becoming functionally complete with kernel version 3.8 (2013), wrap global system resources in abstractions that make it appear to the processes within the namespace that they collectively have access to an isolated instance of the resource [35][36][37][38][39][40]. Namespaces are used for a variety of purposes, the most notable being the implementation of *containers*, a technique for light-weight virtualization. So far, there are six namespace types:

- ◆ *IPC* (`ipc:`, interprocess communication), which isolates System V message queues, semaphore sets, and shared memory segments;
- ◆ *mount* (`mnt:`), which isolates filesystem mount points;
- ◆ *network* (`net:`), which isolates system resources associated with networking — network devices, port numbers, firewall rules, IP (Internet Protocol) addresses, stacks and routing tables, etc.; and
- ◆ *PID* (`pid:`, process ID), which isolates the process ID number space;
- ◆ *user* (`user:`), which isolates the user and group ID number spaces.
- ◆ *UTS* (`uts:`, UNIX Time-sharing System), which isolates the nodename (hostname) and domainname NIS (Network Information Service) system identifiers;

Namespace creation requires privileges, except for user namespaces.

Two namespaces have incidences on the traces captured by LTTng: PID namespaces and user namespaces. It should be noted that namespaces aren't activated in Ubuntu 12.04.3, which has meant that investigation of their consequences on LTTng has had to be postponed.

A process's user and group IDs can be different inside and outside a user namespace (in what follows, whenever user IDs are mentioned, it is understood that group IDs are equally concerned). A process can have a normal unprivileged user ID outside a user namespace while at the same time having a user ID of 0 (root) inside the namespace, giving it full privileges for operations inside the user namespace while remaining unprivileged for operations outside the namespace.

Processes in different PID namespaces can have the same PID. PID namespaces allow containers to be migrated between hosts while keeping the same process IDs for the processes inside the container; they also allow each container to have its own `init` (PID 1, the ancestor of all processes, which manages various system initialization tasks and reaps orphaned child processes when they terminate). PID namespaces can be nested: a process will have one PID for each of the layers of the hierarchy starting from the root PID namespace down to the PID namespace in which it resides. Processes can see (i.e. send signals to) processes contained in their own PID namespace and the namespaces nested inside that PID namespace.

LTTng is already aware of PID namespaces in the sense that it labels process IDs as virtual PIDs (VPIDs, as opposed to the root PIDs). It is not clear yet, however, how the LTTng daemons behave with respect to user namespaces: does the creation of a user namespace potentially entail the spawning of session and consumer daemons local to that namespace, or do the root (a.k.a. real) user-space session and consumer daemons handle any nested user namespaces?

A.6 The `ftrace` facility

`Ftrace` is the kernel's default tracing facility; it originated in real-time enhancement efforts and is now merged into the mainstream [41]. Its feature set depends on the underlying architecture.

Ftrace facility uses kernel instrumentation that adds a systematic call to the `ftrace` manager at nearly every kernel entry point (it can also instrument exit points through stack manipulation). The `ftrace` events are very compact and efficient, noting only the timestamp of the event and the kernel function entered; by default, there is no payload [42] although it does allow payload definition extension [4]. Its focus is on timing and control flow. Like `perf`, `ftrace` is kernel-centric (user-space is traceable with a serious performance hit due to context-switching); it is also strictly single-user. Ftrace includes console output integration that allows dumping tracing buffers upon a kernel crash. Because of its low overhead, `ftrace` is particularly well-suited for tracing high-throughput data coming from frequently hit tracepoints or from function entry/exit instrumentation on busy systems.

Ftrace is controlled by the `/sys/kernel/debug/tracing` folder¹¹, supplied by the `debugfs` file system extension. `Debugfs` may be mounted by default (in `/etc/fstab`) or may need to be mounted manually, using the command “`sudo mount -t debugfs nodev /sys/kernel/debug`”. All “files” within this folder and its subfolders have zero size whatever their contents may be; this is because the contents are actually in the kernel’s memory and not on disk. Instructions are issued to `ftrace` by overwriting or appending to its “files” (typically using the shell’s `>` and `>>` I/O redirectors). `Debugfs` can also control several other tracing facilities such as `kprobe`, `uprobe` and, most notably, LTTng 0.x (development stopped with version 0.89 in May of 2011).

A complete description of how to create, control, and analyse traces generated by the `ftrace` facility is beyond the scope of this document. Support for `ftrace` was removed from LTTng [43] in part because it was felt redundant with the `syscall` facility, and also because of its “global” nature, which is irreconcilable with LTTng’s session semantics. If `ftrace` were to be expanded to support private data, adding it back into LTTng would become possible.

A.7 Updating the system call names

The `lttng-modules/instrumentation/syscalls/README` explains (a little too succinctly) how to update the set of system call LTTng event names whenever the kernel changes. This Topical Note explains the process in more detail.

A kernel source needs to be obtained. Section 4.6.4.1 explains how this is done.

A.7.1 Use `lttng-syscall-extractor`

To make system call name extraction possible, the kernel must be built with the `CONFIG_FTRACE_SYSCALLS` and `CONFIG_KALLSYMS_ALL` options (both are enabled by default). Before (re)building the kernel, apply the `linux-link-trace-syscalls-as-data.patch` linker patch found in `lttng-modules/instrumentation/syscalls/lttng-syscalls-extractor/`. This patch will ensure the kernel keeps the system call metadata in memory after booting.

¹¹ This directory can be visited normally using the command line, but for some reason Ubuntu’s Nautilus will be unable to display its contents. The subfolders (`events`, `options`, `per_cpu`, `trace_stat`) can be displayed normally by Nautilus.

To make sure the patch is applicable, test a dry-run in the kernel source directory:

```
$ cd /home/user/Documents/mykernel-patched/linux-3.9.3  
$ patch -p1 --dry-run -i <path>/linux-link-trace-syscalls-as-data.patch
```

Where `<path>` is the path to the `lttng-syscalls-extractor` patch file (e.g. `/usr/src/lttng-modules/instrumentation/syscalls/lttng-syscalls-extractor`). If the patch checks out, the output will look like this:

```
patching file include/asm-generic/vmlinux.lds.h  
Hunk #1 succeeded at 197 with fuzz 1 (offset 28 lines).  
Hunk #2 succeeded at 517 with fuzz 2 (offset 27 lines).
```

Apply the patch by running the same command without the `dry-run` option. The output will be identical to the `dry-run` case.

Once this is done, the LTTng syscall extractor module (`lttng-syscalls-extractor.c`, same path as `linux-link-trace-syscalls-as-data.patch`) needs to be built. Unfortunately the supplied `Makefile` is incomplete. Append the lines shown in red below:

```
obj-m += lttng-syscalls-extractor.o  
  
all:  
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules  
  
clean:  
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

The `make` lines must be indented using horizontal tabulations (HT, Unicode U+0009) instead of spaces. Now the `lttng-syscalls-extractor` module can be built:

```
$ cd /usr/src/lttng-modules/  
$ cd instrumentation/syscalls/lttng-syscalls-extractor  
$ make
```

See Section 4.6.6 for an example of the kind of expected output. See Section 4.6.4.1 for the procedure to build and install the patched kernel. In summary:

```
$ cd /home/user/Documents/mykernel-patched/linux-3.9.3  
$ make menuconfig  
$ make &> make.log  
$ make modules &> make-modules.log  
$ sudo make modules_install &> install-modules.log  
$ sudo make install &> install.log
```

Reboot into the new kernel. Clear the `dmesg` buffer, then load the module:

```
$ cd /usr/src/lttng-modules/  
$ cd instrumentation/syscalls/lttng-syscalls-extractor  
$ sudo dmesg --clear  
$ sudo insmod ./lttng-syscalls-extractor.ko
```

If you did not reboot into the patched kernel, the `lttng-syscalls-extractor` module will load and the `dmesg` buffer will start with this error:

```
general protection fault: 0000 [#1] SMP
```

If you did reboot into the new kernel, the expected behaviour is for the module to fail to load:

```
insmod: error inserting './lttng-syscalls-extractor.ko': -1
Operation not permitted
```

The `dmesg` output will then hold the system call metadata. Dump it to an appropriately-named file using a command like ‘`sudo dmesg > <outputfile>`’ (examples are found in the kernel-version subdirectories, such as `lttng-modules/instrumentation/syscalls/3.1.0-rc6/x86-32-syscalls-3.1.0-rc6`). Here a 64-bit x86 system is assumed:

```
$ mkdir ..`uname -r`/
$ sudo -E -- sh -c 'dmesg > ..`uname -r`/x86-64-syscalls-`uname -r`'
```

The `syscalls` listing should be something like:

```
[ 201.770696] syscall sys_sched_yield nr 24 nbargs 0 types: () args: ()
[...]
[ 201.771712] SUCCESS
```

A.7.2 Generate the system call `TRACE_EVENT()` macros

Take the `syscalls` listing (`dmesg` metadata) and feed it to `lttng-syscalls-generate-headers.sh`:

```
$ cd /usr/src/lttng-modules/instrumentation/syscalls
$ ./lttng-syscalls-generate-headers.sh integers <input_dir> \
  <input_file> <bitness>
$ ./lttng-syscalls-generate-headers.sh pointers <input_dir> \
  <input_file> <bitness>
```

Where `<input_dir>` is the input sub-directory of `instrumentation/syscalls` (normally `..`uname -r``), `<input_file>` is the name you dumped `dmesg` to (e.g. `x86-64-syscalls-`uname -r``), and the `<bitness>` is either 32 or 64:

```
$ ./lttng-syscalls-generate-headers.sh integers ..`uname -r` \
  x86-64-syscalls-`uname -r` 64
$ ./lttng-syscalls-generate-headers.sh pointers ..`uname -r` \
  x86-64-syscalls-`uname -r` 64
```

Be warned that the current bash script is really dumb and will not give any progress, confirmation or error messages at all (e.g. if using `integer` instead of `integers`). It does not even return an error code when fed incorrect input—it just creates effectively empty headers [44].

If invoked correctly, `lttng-syscalls-generate-headers` will create headers (e.g. `x86-64-syscalls-`uname -r`_integers.h` and `x86-64-syscalls-`uname -r`_pointers.h`) in `/usr/src/lttng-modules/instrumentation/syscalls/headers/`.

The current version of `lttng-syscalls-generate-headers` may generate incorrect headers [44]. At the beginning of the generated header there may be lines like these:

```
[...]
SC_DECLARE_EVENT_CLASS_NOARGS(syscalls_noargs,
    TP_STRUCT_entry(),
    TP_fast_assign(),
    TP_printk()
#ifndef OVERRIDE_64_sys_sched_yield
```

Add the missing new line like this:

```
[...]
SC_DECLARE_EVENT_CLASS_NOARGS(syscalls_noargs,
    TP_STRUCT_entry(),
    TP_fast_assign(),
    TP_printk()
)
#ifndef OVERRIDE_64_sys_sched_yield
```

There may also be an extraneous line at the end of the headers:

```
[...]
SUCCESS

#endif /* CREATE_SYSCALL_TABLE */
```

The `SUCCESS` line should be deleted like this:

```
[...]

#endif /* CREATE_SYSCALL_TABLE */
```

Finally, the new headers can be used to rebuild `lttng-modules` (see Section 3.3.3.1) by editing `instrumentation/syscalls headers/syscalls_integers.h` and `syscalls_pointers.h`. For instance, assuming a 64-bit x86 system and a newly-built 3.13.7 kernel, `syscalls_pointers.h` would be changed from:

```
#ifdef CONFIG_X86_64
#include "x86-64-syscalls-3.10.0-rc7_pointers.h"
#endif
[...]
```

Into (change highlighted in red):

```
#ifdef CONFIG_X86_64
#include "x86-64-syscalls-3.13.7_pointers.h"
#endif
[...]
```

`syscalls_integers.h` would be changed similarly.

A.8 Adding the LTTng PPA to the software sources

A more up-to-date installation of LTTng can be achieved by obtaining the LTTng packages from the LTTng PPA (Personal Package Archive), which is located at <https://launchpad.net/~lttng/+archive/ppa>. As of this writing, adding it to the software sources upgrades the versions from 2.0.1-0ubuntu1 or 2.0.2-0ubuntu1 to 2.3.x+stable for `lttng-modules`, `lttng-tools` and `lttng-ust` (0.6.7-2 to 0.8.x+stable for `liburcu`, 1.0.0~rc1 to 1.0.x+stable for `babeltrace`).

The LTTng PPA offers an additional package, `lttngtop`. This is one of several approaches to analysing the traces produced by LTTng (others are Eclipse's Tracing and Monitoring Framework (TMF) and RapidMiner's LTTng plug-ins). You may optionally install it along with LTTng proper. `Lttngtop` can also be installed from the `git.lttng.org` repository: see Section 3.3.3.6.

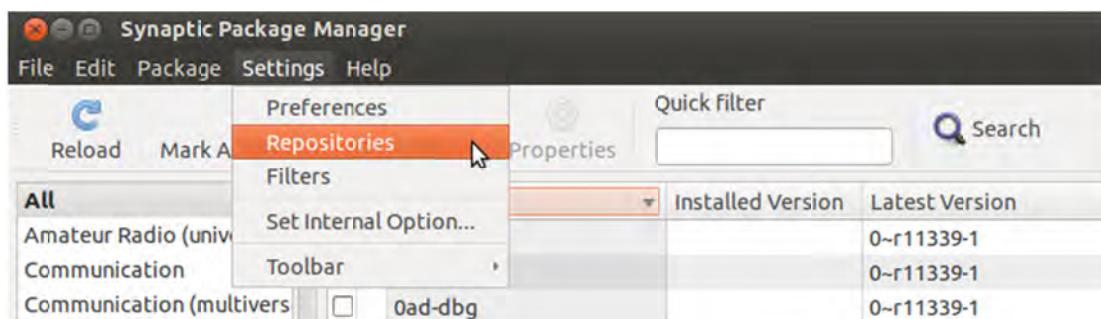
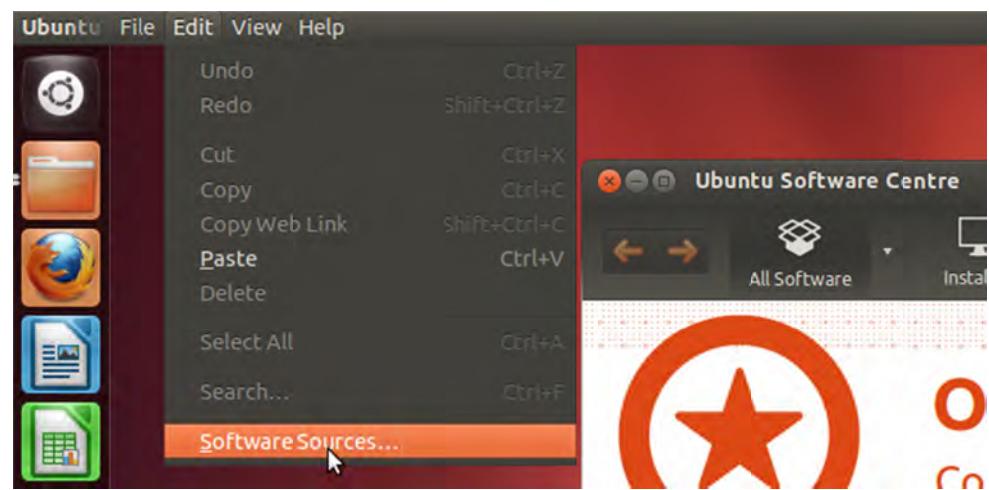


Figure 37: Selecting the software sources.
Top: from the Ubuntu Software Centre; bottom: from the Synaptic Package Manager.

To access the LTTng PPA from the Ubuntu Software Centre, select “Edit: Software Sources” (Figure 37, top). From the Synaptic Package Manager, select “Settings: Repositories” instead (Figure 37, bottom). Both commands open the same window. Go to the “Other Software” tab (Figure 38), click “Add...” and specify the following APT line (Ubuntu 12 is “Precise Pangolin”):

```
deb http://ppa.launchpad.net/lttng/ppa/ubuntu precise main
```

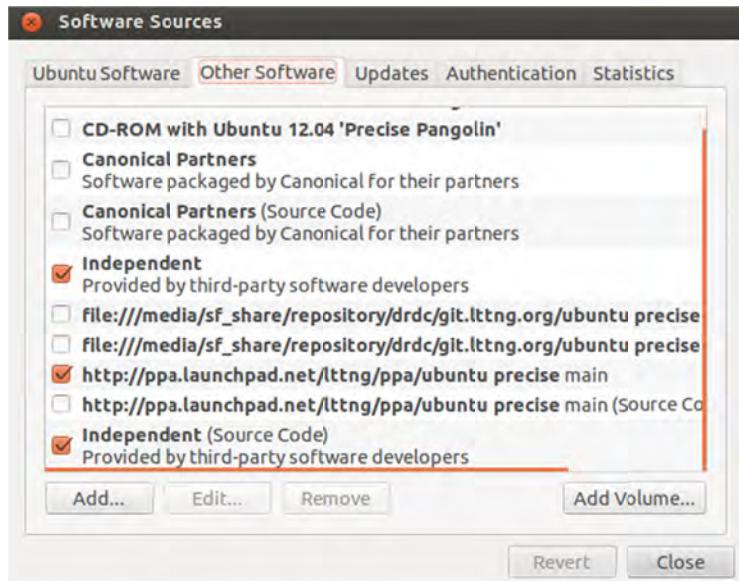


Figure 38: The Other Software tab of the Software Sources.

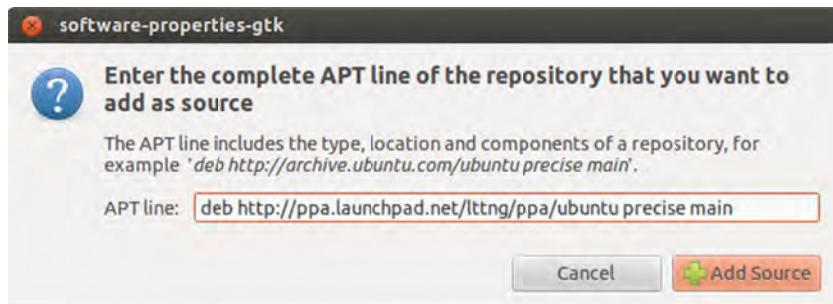


Figure 39: Adding a software source.

Click “Add Source” (Figure 39), then “Close” (Figure 38), then Synaptic’s “Reload” (Figure 37, bottom). The Ubuntu Software Centre may need to be closed and re-opened to force a refresh.

To install the PPA from the command line, you do this instead:

```
$ sudo add-apt-repository ppa:lttng/ppa
$ sudo apt-get update
```

Once the PPA has been added to the software sources, install LTTng as described in Section 3.1.

DRDC users may run into trouble because of the corporate firewall, particularly if using the LTTng PPA (because it is not a trusted source yet). If the install stalls, follow the instructions in Topical Note A.15, *DRDC firewall palliation*.

A.8.1 Getting the PPA's authentication key

You will also need the PPA's PGP key if you want to avoid annoying warnings about untrusted sources. Installation will work without this step, but more user intervention will be required. The `add-apt-repository` command line used above includes fetching the key, so the following only applies when using the GUI.

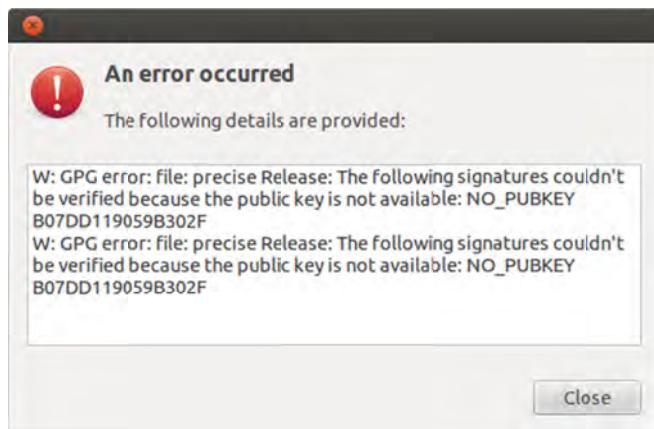


Figure 40: The `NO_PUBKEY` error message.

You must first get the PGP key's ID or its number. If you add the repository and then reload Synaptic, you will get a `NO_PUBKEY` GPG error (see Figure 40 above) that lists the public key number (e.g. `B07DD119059B302F` for the DVD-ROM's software repository); the ID is just the lower half of this (e.g. `059B302F`). Note this number and skip to Section A.8.2.

You can also go to any PGP public key server (`keyserver.ubuntu.com`, `pgpkeys.mit.edu` (also known as `pgp.mit.edu`), `pgp.openpkg.org`, `sks.pkgs.net`, `pool.sks-keyservers.net`, `zimmermann.mayfirst.org`, etc.) and enter the key owner's name as a search string. The DVD-ROM's software repository is signed by the author of this document, so you'd search for `daniel.thibault@drdc-rddc.gc.ca`. You should get results similar to those shown in Figure 41 below.

Search results for 'thibault rddc gc drdc daniel ca'

Type	bits/keyID	cr. time	exp time	key expir
pub	2048R/ 059B302F	2013-05-09		
	Fingerprint=FBB1 7720 4C69 8771 40FD 3122 B07D D119 059B 302F			
uid	Daniel U. Thibault < daniel.thibault@rddc.gc.ca >			
sig	sig3	059B302F	2013-05-09	[selfsig]
sub	2048R/E789DFF9	2013-05-09		
sig	sbbind	059B302F	2013-05-09	[1]

Figure 41: A PGP Public Key Server search results screen.

Clicking on the key ID will display the PGP public key block (see Figure 45).

PPA Web pages also list their signing keys. Go to the LTTng PPA Web page given earlier (Figure 42) and click on “Technical details about this PPA” to deploy its text (Figure 43).

The screenshot shows the LTTng Stable PPA page. At the top, there's a logo for the "Ubuntu LTTng" team featuring three stylized human figures. The main title is "Ubuntu LTTng" team with "Ubuntu LTTng" in quotes. Below the title, there's a navigation bar with links for "Overview", "Code", "Bugs", "Blueprints", "Translations", and "Answers". The main content area has a heading "LTTng Stable PPA". Below it, a link says "Ubuntu LTTng team » LTTng Stable PPA". The "PPA description" section contains the following text:
LTTng (Linux Trace Toolkit Next Generation) project repository.
For more information: <http://lttng.org>
Most of the LTTng utilities are now packaged in Debian/Ubuntu, but this PPA offers the latest stable versions. Only Ubuntu 12.04 and up are supported.

Figure 42: The LTTng PPA's Web page.

Adding this PPA to your system

You can update your system with unsupported packages from this untrusted PPA by adding [ppa:lttng/ppa](#) to your system's Software Sources. ([Read about installing](#))

▽ Technical details about this PPA

This PPA can be added to your system manually by copying the lines below and adding them to your system's software sources.

Display sources.list entries for: [Choose your Ubuntu version](#) ▾

```
deb http://ppa.launchpad.net/lttng/ppa/ubuntu
YOUR_UBUNTU_VERSION_HERE main
deb-src http://ppa.launchpad.net/lttng/ppa/ubuntu
YOUR_UBUNTU_VERSION_HERE main
```

Signing key:

[1024R/33739778 \(What is this?\)](#)

Fingerprint:

C541B13BD43FA44A287E4161F4A7DFFC33739778

For questions and bugs with software in this PPA please contact  [Ubuntu LTTng](#).

Figure 43: The LTTng PPA's technical details.

Click on the signing key link (“1024R/33739778”); this leads to a page giving the search results for the key (Figure 44). Click on the keyID (“33739778”) to display the public key block (Figure 45).

Search results for '0xc541b13bd43fa44a287e4161f4a7dfffc33739778'

Type	bits/keyID	Date	User ID
pub	1024R/33739778	2010-02-09	Launchpad lttng-ppa

Figure 44: Key search results.

Public Key Server -- Get ``0xf4a7dfffc33739778 ''

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: SKS 1.1.4
Comment: Hostname: keyserver.ubuntu.com

mI0ES3G/SQEEALatOVK/FRIAueWdHJpFetF6afJENb619/eVmxEHirddYVO6vSxxOJ9XZYbN
zaIA9nF039Hdmz6jkmH4bDpJ5bT38YqJHfWGPO6X2nAD6F05XILKghSLovDU67XRtSxKTJM
giqkw3DL1ss+XY22Q3spSFFOStxb2WjC5rcyW2pFABEBAAQOE0xhdW5jaHBhZCBsdHRu2y1w
cGGItgQTAQIAIAUCSSG/SQIBAwYLQgHAwIEFQIIAwQNAgMBAh4BAheAAAoJEP$sn3/wzc5d4
QoKD/1Ny4IGUZGRLje4c3XM7ZxdCd5dvX9nDEUmF3tk6+SO+AmynLRISV61cSBPH79kNLS4B
XGx2zTkvxChmVtz8JYFmxTILo2LXBUCmlETA/HjtZK3w+XRIEbFfLC7buAHIZJSrUca0JcsU
na3ZPv9K1ExsoZe2aHqCkr8/ANvOMqe+
=cLqe
-----END PGP PUBLIC KEY BLOCK-----
```

Figure 45: Key ID.

Select the PGP public key block, including the lines that read “BEGIN PGP PUBLIC KEY BLOCK” and “END PGP PUBLIC KEY BLOCK”, and copy that to a blank text document (using gedit, for instance). Save this as “LTTng PPA public key.asc” or some such. This is known as an “ASCII Armored file” in PGP parlance.

Now go to Synaptic’s Settings: Repositories: Authentication (or the Ubuntu Software Centre’s Edit: Software Sources: Authentication) (Figure 46), click Import Key File..., and select the .asc file you saved earlier (for some users, this will not work, as reported in [45]). The equivalent command line for this last step is:

```
$ sudo apt-key add armoured_key_file.asc
```

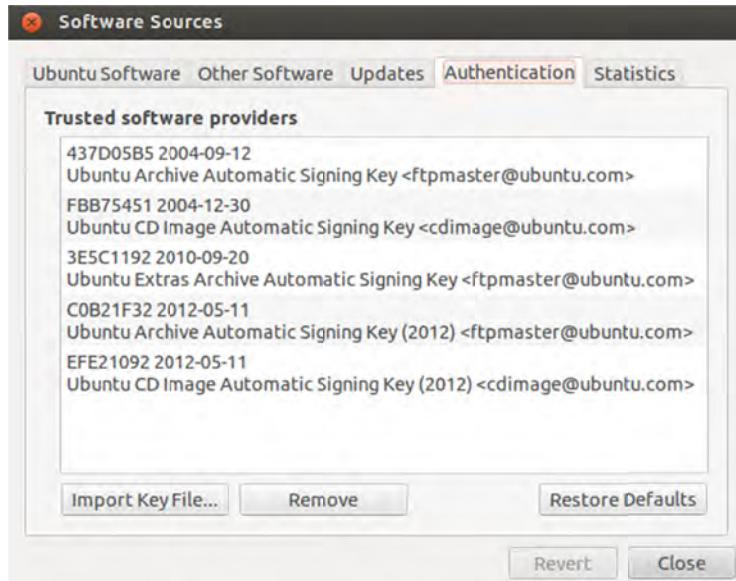


Figure 46: The Software Sources Authentication dialog.

A.8.2 Receiving a PGP key directly from a server

The command lines to get the key (starting from a NO_PUBKEY error message) and add it to Synaptic without the intermediate step of saving it to file is:

```
$ gpg --keyserver pgpkeys.mit.edu --recv-key F4A7DFFC33739778
$ gpg --export --armor F4A7DFFC33739778 | sudo apt-key add -
```

Where F4A7DFFC33739778 is the public key number reported by the NO_PUBKEY error message (the last 16 hexadecimal digits of the fingerprint reported in the PPA’s technical details).

The list of “Trusted Software Providers” should now include “Launchpad lttng-ppa”. After closing the Repositories dialog, click Reload to refresh Synaptic’s list of packages (equivalently, issue the sudo apt-get update command).

Currently, for some PGP key servers DRDC users will be blocked by the corporate firewall at the key search results step (Figure 44) or, if using the `gpg --keyserver` command line, they will get a “Connection timed out” HTTP fetch error 7. The ASCII Armored file will need to be obtained using an external work station and e-mailed to the DREnet work station. Try `keyserver.ubuntu.com` first: it seems to be more reliably accessible through the DRDC firewall.

A.8.3 Downloading packages from the LTTng PPA

You can also download any LTTng package directly from the PPA’s Web interface. From the starting page, find the “View package details” link, just below the “Technical Details” (Figure 47).

The screenshot shows the "PPA statistics" section of the LTTng PPA. It includes a "Technical details about this PPA" link and a "View package details" link, which is circled in red. Below the main content, there's a note about contacting Ubuntu LTtng for questions and bugs.

Figure 47: The View Package Details link.

Click it to reach the Package Details (Figure 48). Find the package of interest based on its name (babeltrace, liburcu, etc.) and the series matching your system (in the Ubuntu 12.04.3 LTS case, precise).

The screenshot shows a table of package details. The columns are: Source, Uploader, Published, Status, Series, Section, and Build Status. The table lists several packages, including babeltrace and liburcu, across different versions and series (Raring, Precise, Quantal, Saucy).

Source	Uploader	Published	Status	Series	Section	Build Status
▷ babeltrace - 1.0.x+stable-0+bzr634+pack18+201303011559-raring1	(changes file)	no signer	2013-03-01	Published	Raring	Libs ✓
▷ babeltrace - 1.0.x+stable-0+bzr634+pack18+201303011559-precise1	(changes file)	no signer	2013-03-01	Published	Precise	Libs ✓
▷ babeltrace - 1.0.x+stable-0+bzr634+pack18+201303011558-quantal1	(changes file)	no signer	2013-03-01	Published	Quantal	Libs ✓
▷ liburcu - 0.7.x+stable-0+bzr824+pack20+201305101744-saucy1	(changes file)	no signer	1 hour ago	Published	Saucy	Libs ✓
▷ liburcu - 0.7.x+stable-0+bzr824+pack20+201305101743-raring1	(changes file)	no signer	1 hour ago	Published	Raring	Libs ✓
▷ liburcu - 0.7.x+stable-0+bzr824+pack20+201305101743-quantal1	(changes file)	no signer	1 hour ago	Published	Quantal	Libs ✓
▷ liburcu - 0.7.x+stable-0+bzr824+pack20+201305101743-precise1	(changes file)	no signer	1 hour ago	Published	Precise	Libs ✓

Figure 48: The Package Details.

Clicking each package reveals its details, including the package files that make it up (Figure 49).

Publishing details
 Built by recipe [ltng-tools-stable](#) for Ubuntu LTng
 Published on 2013-05-06

Changelog
`ltng-tools (2.1.x+stable-0+bzr1365+pack4+201305061822~precise1) precise; urgency=low`
 * Auto build.
 -- Ubuntu LTng <ltng@lists.launchpad.net> Mon, 06 May 2013 18:22:07 +0000

Available diffs
[diff from 2.1.x+stable-0+bzr1364+pack4+201304232149~precise1 to 2.1.x+stable-0+bzr1365+pack4+201305061822~precise1](#) (865 bytes)

Builds
 amd64
 i386

Built packages
[libltng-ctl-dev](#) LTng control and utility library (development files)
[libltng-ctl0](#) LTng control and utility library
[ltng-tools](#) LTng control and utility programs

Package files
[libltng-ctl-dev_2.1.x+stable-0+bzr1365+pack4+201305061822~precise1_amd64.deb](#) (75.4 kB)
[libltng-ctl-dev_2.1.x+stable-0+bzr1365+pack4+201305061822~precise1_i386.deb](#) (76.4 kB)
[libltng-ctl0_2.1.x+stable-0+bzr1365+pack4+201305061822~precise1_amd64.deb](#) (55.6 kB)
[libltng-ctl0_2.1.x+stable-0+bzr1365+pack4+201305061822~precise1_i386.deb](#) (55.1 kB)
[ltng-tools_2.1.x+stable-0+bzr1365+pack4+201305061822~precise1.dsc](#) (965 bytes)
[ltng-tools_2.1.x+stable-0+bzr1365+pack4+201305061822~precise1.tar.gz](#) (395.2 kB)
[ltng-tools_2.1.x+stable-0+bzr1365+pack4+201305061822~precise1_amd64.deb](#) (231.3 kB)
[ltng-tools_2.1.x+stable-0+bzr1365+pack4+201305061822~precise1_i386.deb](#) (224.9 kB)

Figure 49: A package's Details.
 The package file links are at the bottom.

You can download the binary packages relevant to your architecture (e.g. the *_amd64.deb packages for a 64-bit machine) as well as the source code (the tar.gz file and accompanying .dsc description file). Debian packages may be installed using the Software Centre as mentioned in Section A.2. You can also open any .deb package file with the Archive Manager and extract the files you need to a staging folder of your choice. The GUI's contextual menu offers an “Extract Here” entry which will deploy the package's entire contents as a subfolder of the package's.

A.9 Java/JNI support

If you want LTTng to offer user-space Java/JNI support, you'll of course have to have a JDK installed (such as the `openjdk-7-jdk` package). The `configure` call of the `lttng-ust` tarball will also need to receive a couple of extra options (this is oddly omitted from `lttng-ust`'s `README`):

```
$ ./configure --with-java-jdk=/usr/lib/jvm/<jdk> \
--with-jni-interface &> configure.log
```

Where `<jdk>` is the particular Java virtual machine you have on your system. For `openjdk-7-jdk`, this will be `java-7-openjdk-amd64`. With Java/JNI support turned on, the `libltng-ust-java` subfolder will be processed; otherwise it is ignored by `make`.

Since no Linux kernel uses any Java, there is no comparable option in any of the other LTTng tarballs.

A.10 SystemTap support

SystemTap suffers from a similar omission in `lttng-ust`'s README: to integrate user-space SystemTap output, install the `systemtap-sdt-dev` package and add `with-sdt` to the `configure` options. SystemTap itself (the `systemtap` package) can be installed later. The support allows SystemTap output to be captured in the LTTng trace.

A.11 The GNU gold bug

The `binutils-gold` package (the `gold` linker) was listed as required by `lttng-tools` up to and including the 2.1.1 release; this requirement was relaxed with the 2.2.0 release. You thus may or may not be using it (it is described as “a new linker, which is faster than the current linker included in `binutils`”). If you are using `gold`, you will have to prefix each `./configure` call of the `lttng` toolchain that depends on other parts of the toolchain (i.e. `lttng-ust` and `lttng-tools`) with `LDFLAGS=-L/usr/local/lib`. Otherwise, `configure` or `make` will fail with a “Cannot find” error. Like so:

```
$ LDFLAGS=-L/usr/local/lib ./configure &> configure.log
```

According to the `ld-linux` man pages, the Linux default library/header paths are just `/lib` and `/usr/lib`: they do *not* include `/usr/local/lib`. However, `/etc/ld.so.conf.d/libc.conf` (installed by the Ubuntu `libc-bin` package) adds `/usr/local/bin` to the system loader's path, so any application that relies on `/usr/local/bin` libraries will find them. It turns out this particular failure of `gold` is a known bug [46]: unlike the linker it replaces, the `gold` linker does not look into `/usr/local/lib` by default and ignores the `ld.so.conf` instructions. This old `gold` bug may eventually be fixed.

Another way to resolve these problems is to issue the `prefix=/usr` option to `configure` (in all cases except the `lttng-modules` tarball, which does not have a `configure` step); this means the installation (`make install`) will target the `/usr/lib` folder instead of `/usr/local/lib`, making the `lttng` libraries globally visible.

A.12 Setting compilation flags

Sometimes a tarball's README will instruct you to enable a certain compilation flag in order to achieve some effect. For instance, “the compilation flag `"-DLTTNG_UST_DEBUG_VALGRIND"` should be enabled at build time to allow `liblttng-ust` to be used with `valgrind` (side-effect: disables per-CPU buffering).” [47]

To enable a compilation flag, you either modify the `Makefile`'s target `$(CC)` line to add the '`-D<flag>`' or '`-D <flag>`', or, if you'd rather not tamper with the `Makefile`, you invoke '`make FLAG="-D<flag>" [...]`' (here `<flag>` could be `LTTNG_UST_DEBUG_VALGRIND`, for example). You can also set this up upstream, by modifying the `CFLAGS` of your `Makefile` in the `configure` call (this will depend on the specific variables used in the `Makefile`; you'll probably do something like `./configure CFLAGS="-I. -D <flag>"`). Conversely (downstream), you can set the `#define <flag>` explicitly in the appropriate source file.

A.13 The Linux dynamic loader

When an executable is to be launched, the dynamic loader (`ld.so`) must complete its linking so it can become a process. The executable's description lists the libraries that it depends on to run. These libraries may, in turn, have dependencies of their own. If any of these dependencies are not satisfied (i.e., if the required libraries cannot be found), the process cannot run. Precisely the same thing occurs in order to satisfy an explicit library load command (the `dlopen()` system call).

A.13.1 The library search path

The dynamic loader searches for the requested library in the following ordered list of places, the *library search path*, stopping as soon as a match is found:

- the `rpath` folder (or colon-separated folders) encoded in the requesting executable, if any, but only if `runpath` is not also set (e.g. `gcc -Wl,-rpath,'$ORIGIN/rpath' <etc.>`. `$ORIGIN` is the executable's current location; to avoid substitution problems you must either single-quote an `$ORIGIN` path or escape the leading `$`; instead of using the linker `-rpath` option, you can set `LD_RUN_PATH` before calling the linker);
- the (colon-separated) folders listed in the environment variable `LD_LIBRARY_PATH`, if any;
- the `runpath` folder (or colon-separated folders) encoded in the executable, if any (e.g. `gcc -Wl,-rpath,'$ORIGIN/rpath',--enable-new-dtags <etc.>`; although it is possible to modify the binary so `rpath` and `runpath` are different, `gcc` sets both values to the same string);
- the folders listed or included in `/etc/ld.so.conf`, if any;
- the trusted folder `/lib`; and finally
- the trusted folder `/usr/lib`.

The last three locations are known as the “default library locations”; if an executable is linked with the `-z nodeflib` option (“no def[ault] lib[raries]”), then `ld.so` will not look in those locations when servicing requests from that executable (that is to say, the search will stop with the third bullet above). The Linux `/etc/ld.so.conf` typically includes (i.e. there is an explicit `include` instruction in `/etc/ld.so.conf`) the collection of `.conf` files in the `/etc/ld.so.conf.d` folder. The `/etc/ld.so.conf.d` files themselves are sorted alphabetically. Finally, it should be noted that the dynamic loader does not actually scan `/etc/ld.so.conf` but instead uses a compiled version of it, `/etc/ld.so.cache`, which the `ldconfig` command updates upon request. A stale `ld.so.cache` is a frequent cause of apparent errors.

There are some additional caveats to the search path. For security reasons, so-called set-user-ID/set-group-ID (setuid/setgid) binaries ignore the `LD_LIBRARY_PATH`. Also, preload (see `LD_PRELOAD` and `/etc/ld.so.preload`, below) pathnames containing slashes are ignored, and libraries in the standard search folders are loaded only if the set-user-ID permission bit is enabled on the library file. Old `a.out` format binaries also use their own versions of `LD_LIBRARY_PATH` and `LD_PRELOAD`, named `LD_AOUT_LIBRARY_PATH` and `LD_AOUT_PRELOAD`, respectively.

You can find out what the `rpath` and `runpath` tags of an executable are with “`readelf --dynamic <executable> | grep PATH`”.

A.13.2 The libraries sought

Usually, the command line starts the dynamic loader on just one object, the executable, previously sought through the run search path. The details of the latter won’t be discussed here. However:

- the libraries (*not* folders) listed in the environment variable `LD_PRELOAD`, if any (absolute or relative paths; `LD_PRELOAD` accepts colon- and/or space-separated entries), are added to the top-level list of sought libraries; and
- the libraries (*not* folders) listed in the system file `/etc/ld.so.preload`, if any, are also added to the list.

For a command line such as `LD_PRELOAD=one two`, the dynamic loader’s starting request is for `{two, one}`. The search is conducted breadth-first, and continues recursively until all dependencies are satisfied or the process fails because a library isn’t found. Thus the dependencies of each top level module are added to the list of modules to load, then the dependencies of these additions are added to the list, and so on. Reoccurrences of libraries are dropped from the list as they are encountered, of course. Figure 50, below, illustrates this process.

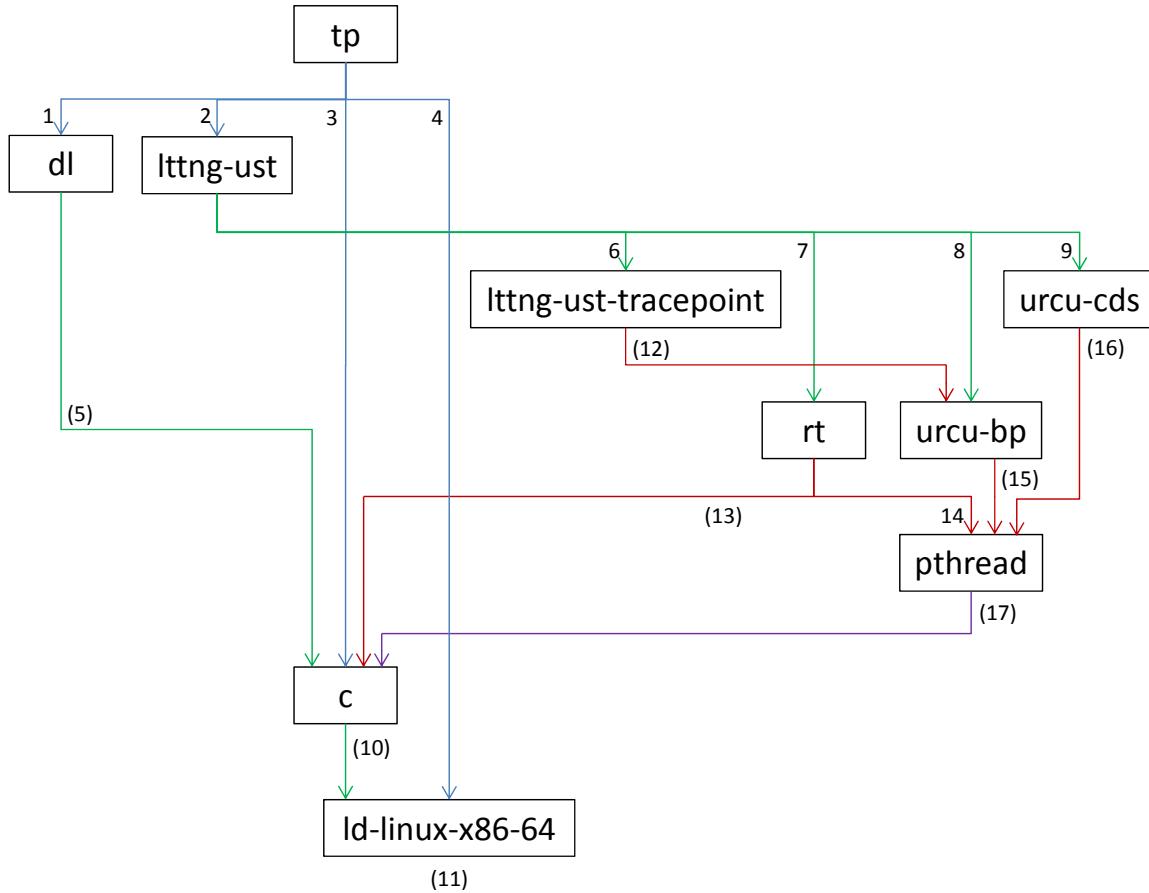


Figure 50: The dependency tree of a basic tracepoint provider.

Successive traversals are colour-coded: **blue** for the first pass, **green** for the second, **red** for the third, and **violet** for the fourth. Searches are numbered in sequence, and those already resolved by the time they arise (and are thus skipped) are in parentheses.

A.13.3 Initializations and finalizations

Once all libraries are found, they are initialised in reverse loading order (i.e. the libraries are pushed on a stack as they are found). The loader (`ld`) self-initializes first, out of order, with the exception of `pthread` which initializes before the loader when it is present. In the example of Figure 50, the loading order is `{ tp, dl, lttng-ust, c, ld, lttng-ust-tracepoint, rt, urcu-bp, urcu-cds, pthread }`. The initialization order is `{ pthread, ld, urcu-cds, urcu-bp, rt, lttng-ust-tracepoint, c, lttng-ust, dl, tp }`.

Note that if `LD_PRELOAD` or `/etc/ld.so.preload` supplied additional libraries, these are initialised after the libraries they replace, even if they are true repeats (for instance, `LD_PRELOAD=/usr/bin/man /usr/bin/man` will initialize the `man` program and then call `init` on `man` again). The preloaded library members (functions and objects) pre-empt those from the previously found library. Preloading thus serves to substitute replacements for some of the “normal” library methods (and yes, preloading a full replacement does mean the replaced library will have been initialised for naught). In the full dynamicity case (using an explicit `dlopen()` call to load a library), the library search and load occurs *after* the executable has been loaded, initialised and started. However, if the library was preloaded, it will have been initialised before the executable, and `dlopen` will find the preloaded library.

Once the process concludes, it calls its finalizers in the reverse order of initialization. Exceptionally, a finalizer may itself cause a library to be sought, loaded, initialized and later finalized.

A.13.4 Making libraries findable

`LD_LIBRARY_PATH` is widely considered broken, because it prefixes the search path for *all* executables in the environment, making search conflicts likely; it also undercuts any `rpath` set in the executable (fixing this was the motivation behind adding the `runpath` tag). Thus, setting the `LD_LIBRARY_PATH` permanently is not recommended. Setting `rpath` or `runpath` to an absolute path is also a bad idea since it constrains where the executable’s libraries can be installed.

A.14 Kernel instrumented module templates and scripts

The following is a template for an LTTng kernel tracepoint handler module source code file (`lttng-probe-<system>.c`, repeated from Section 4.6.2):

```
#include <linux/module.h>

/*
 * Create the tracepoint static inlines from the kernel to validate
 * that the trace event macros match the kernel we run on.
 */
#include <trace/events/<system>.h>

#include "../wrapper/tracepoint.h"

/*
 * Create LTTng tracepoint probes.
 */
#define LTTNG_PACKAGE_BUILD
#define CREATE_TRACE_POINTS
#define TRACE_INCLUDE_PATH ../instrumentation/events/lttng-module

#include "../instrumentation/events/lttng-module/<system>.h"

MODULE_LICENSE("GPL and additional rights");
MODULE_AUTHOR("Daniel U. Thibault <daniel.thibault@drdc-rddc.gc.ca>");
MODULE_DESCRIPTION("LTTng <system> probes");
```

Here is the kernel tracepoint header template (`<system>.h`) you must use with the script given later (the script adds the leading `#define` statements that appear in the examples given in Section 4.6.6):

```
//The LTTng tracepoint provider kernel module will be
//named 'lttng-probe-TRACE_SYSTEM'
#undef TRACE_SYSTEM
#define TRACE_SYSTEM <system>

#if !defined(_TRACE_<SYSTEM>_H) || defined(TRACE_HEADER_MULTI_READ)
#define _TRACE_<SYSTEM>_H

#include <linux/tracepoint.h>
```

```

#ifndef _TRACE_<SYSTEM>_DEF
#define _TRACE_<SYSTEM>_DEF
//Declare consts, enums, working structs, helper functions here
#endif /* _TRACE_<SYSTEM>_DEF */

DECLARE_EVENT_CLASS(<template>,
    TP_PROTO(<declarator's parameter type list>),
    TP_ARGS(<corresponding identifier list>),
    TP_STRUCT_entry(
        __field(<type>, <field>)
        __array(<type>, <field>, <size>)
        dynamic_array(<type>, <field>, <size>)
        string(<field>, <string>)
    ),
    TP_fast_assign(
#ifdef LTTNG_MAINLINE
    __entry-><field> = <expr>;
    memcpy(__entry-><field>, <expr>, <size in bytes>));
    memcpy(__get_dynamic_array(<field>), <expr>, <size in bytes>));
    assign_str(<field>, <expr>);
#else
    tp_assign(<field>, <expr>)
    tp_memcpy(<field>, <expr>, <size in bytes>)
    tp_memcpy_dyn(<field>, <expr>)
    tp_strcpy(<field>, <expr>)
#endif
),
    TP_printk(
        //Use __entry-><f> for scalar or static array <f>
        //      __get_str(<f>) for __string(<f>)
        //      __get_dynamic_array(<f>) for __dynamic_array(<type>, <f>)
        //You can use "%s", __entry-><f> when <f> is __array(char...
        "%d %s", <expr>, <expr>
    )
) _LTTNG_SEPARATOR

DEFINE_EVENT(<template>, <event name>,
    TP_PROTO(<matching the template's>),
    TP_ARGS(<matching the template's>)
) _LTTNG_SEPARATOR

TRACE_EVENT(<event name>,
    <remainder as per DECLARE_EVENT_CLASS>
) _LTTNG_SEPARATOR

```

```

// See also TRACE_EVENT_CONDITION, DEFINE_EVENT_CONDITION,
// DEFINE_EVENT_PRINT
// lttng-module only: TRACE_EVENT_MAP, TRACE_EVENT_CONDITION_MAP
// DEFINE_EVENT_MAP, DEFINE_EVENT_PRINT_MAP, DEFINE_EVENT_CONDITION_MAP

#endif /* _TRACE_<SYSTEM>_H */

/* This part must be outside protection */
#undef _LTTNG_SEPARATOR
#ifndef _LTTNG_MAINLINE
#include <trace/define_trace.h>
#else
#include "../../probes/define_trace.h"
#endif

```

The module source code file (`<system>.c`) is assumed to already exist in the working directory (see Section 4.6.6 for a sample). The script assumes the same title (`hello` in Section 4.6.6) is used for the header and the (optional) custom module; this is of course an imperfect assumption. The LTTng kernel tracepoint provider module source code (`lttng-probe-<system>.c`) is also assumed to already exist in the working directory. The one thing the script does not do is add `lttng-probe-<system>` to `lttng-modules/probes/Makefile`, but it does check if this has been done.

The module source code file (`<system>.c`) need not exist: if absent, such as when instrumenting a custom kernel (or its modules) directly, the custom module is not built. See Section 4.6.4.1 for help in building and installing a custom kernel.

The script (`process-module`) does away with the need to create a separate `lttng-module` version of `<system>.h` and handles all the copying and making for you. A reasonable effort was made to make it as foolproof as possible.

```

#!/bin/bash
#
# Copyright © 2014 - Daniel U. Thibault
#                      <daniel.thibault@drdc-rddc.gc.ca>
#
# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public License
# as published by the Free Software Foundation; only version 2
# of the License.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
#
# This script takes one argument, an LTTng kernel tracepoint header
# title (i.e. without the trailing .h; basically the TRACE_SYSTEM
# #define).
#
# The custom module to instrument is assumed to be have the same title
# but the .c extension. If not found, a warning is issued but
# processing goes ahead nevertheless. An alternate name can be supplied
# as the script's second argument.
#
# The script first copies lttnng-probe-$1.c to lttnng-modules/probes,
# then it prepends a couple of #define lines to $1.h and copies it to
# /usr/src/linux-headers-`uname -r`/include/trace/events and
# lttnng-modules/instrumentation/events/mainline (the latter as a
# symbolic link to the former).
#
# The script checks whether lttnng-probe-$1 has been added to
# lttnng-modules/probes/Makefile.
#
# linux-source is used if linux-headers isn't found. A custom kernel
# source can be specified by prefixing the script command like so:
#
# LINUXHEADERS=~/Documents/mykernel/linux-3.9.3 ./process-module hello
#
# Finally, the module's make is invoked (if there was a module source),
# then lttnng-modules is re-built and re-installed. The makes and
# installs are logged.
#
# Daniel U. Thibault, 2014-Mar-03
#

```

```

# Exit if something goes wrong
set -e

# Do we have too many or too few arguments?
if [ $# -gt 2 ] || [ $# -lt 1 ]; then
    echo "Error: Too many or too few arguments. The syntax is:"
    echo "process-lttng-system <system> [<instrumented-module>]"
    exit 1
fi

# We need to export $1 to a sub-shell later on
# Default module source title to system title
TRACE_SYSTEM=$1
if [ $# -eq 2 ]; then
    THE_MODULE=$2
else
    THE_MODULE=$TRACE_SYSTEM
fi

# Is the module source present?
if [ ! -f "./$THE_MODULE.c" ]; then
    echo "Warning: No instrumented module source specified or found."
    THE_MODULE=
else
    echo "$THE_MODULE.c found"
fi

echo "~~~~~Checking for $TRACE_SYSTEM.h and lttng-probe-$TRACE_SYSTEM.c"
# Is the LTTng kernel tracepoint header present?
[ ! -f "./$TRACE_SYSTEM.h" ] && \
echo "Error: File ./${TRACE_SYSTEM}.h (the kernel tracepoint header) " \
    "not found." \
&& exit 1
echo "$TRACE_SYSTEM.h found"

# Is the LTTng kernel tracepoint provider module source present?
[ ! -f "./lttng-probe-$TRACE_SYSTEM.c" ] && \
echo "Error: File ./lttng-probe-$TRACE_SYSTEM.c (the LTTng" \
    "kernel tracepoint provider module source) not found." \
&& exit 1
echo "lttng-probe-$TRACE_SYSTEM.c found"

```

```

echo "~~~~~Checking for lttng-modules"
# Can the lttng-modules directory be found?
# The '|| [ -z "" ]' is to force the statement to succeed
# (otherwise the script would exit immediately)
# Note: We shut stderr up (2>&-) because the "normal" failure emits
# "Package `lttng-modules' is not installed. [...]"
LTTNGMODULES=`dpkg --listfiles lttng-modules 2>&- | \
    grep "lttng-modules[^/]*$"` || [ -z "" ]
[ -z "$LTTNGMODULES" ] && \
echo "Error: lttng-modules is not installed." \
&& exit 1
[ ! -d "$LTTNGMODULES" ] && \
echo "Error: The $LTTNGMODULES directory is missing;" \
    "please re-install lttng-modules." \
&& exit 1
echo "lttng-modules found in $LTTNGMODULES"

# Trim trailing "/" (added by command-line completion)
LINUXHEADERS=${LINUXHEADERS%/>
if [ ! -z "$LINUXHEADERS" ]; then
    echo "~~~~~Validating custom linux-headers path"
    [ ! -d "$LINUXHEADERS/include/trace/events" ] && \
    echo "Error: The $LINUXHEADERS/include/trace/events" \
        "directory is missing." \
    && exit 1
# Extract the title
    LINUXHEADERS_NAME=${LINUXHEADERS##*/}
    HEADERS_OR_SOURCE=custom
    echo "$LINUXHEADERS/include/trace/events found"
else
    echo "~~~~~Checking for linux-headers"
    HEADERS_OR_SOURCE=linux-headers
# Can the linux-headers directory be found (for the current kernel)?
    LINUXHEADERS_NAME=linux-headers-`uname -r`
    LINUXHEADERS=`dpkg --listfiles $LINUXHEADERS_NAME 2>&- | \
        grep "src/linux-headers[^/]*$"` || [ -z "" ]
    if [ -z "$LINUXHEADERS" ]; then
# No linux-headers for the current kernel; try linux-source
        echo "~~~~~Falling back on linux-source"
        HEADERS_OR_SOURCE=linux-source
        LINUXHEADERS_NAME=`uname -r`
# Strip release and qualifiers (e.g. 3.2.0-54-virtual becomes 3.2.0)
# (because kernel.org and Ubuntu only distribute the latter)
        LINUXHEADERS_NAME=linux-source-${LINUXHEADERS_NAME%*-}
        LINUXHEADERS=`dpkg --listfiles $LINUXHEADERS_NAME 2>&- | \
            grep "src/linux-source[^/]*$"` || [ -z "" ]
    fi

```

```

[ -z "$LINUXHEADERS" ] && \
echo "Error: No linux-headers or linux-source found for the" \
      "current kernel. Please install either" \
      "linux-headers-`uname -r` or $LINUXHEADERS_NAME." \
&& exit 1
[ ! -d "$LINUXHEADERS" ] && \
echo "Error: The $LINUXHEADERS directory is missing;" \
      "please re-install $LINUXHEADERS_NAME." \
&& exit 1
echo "$LINUXHEADERS found"

echo "~~~~~Checking for collision between $HEADERS_OR_SOURCE" \
      "and $TRACE_SYSTEM"

# Make sure the target in HEADERS_OR_SOURCE is not a distribution
# header (i.e. don't overwrite standard headers). We cannot just check
# to see if $LINUXHEADERS/include/trace/events/$TRACE_SYSTEM.h exists
# because it will after the script has been run once. We need to find
# out whether the target .h pre-existed or not, i.e. whether it came
# from a $HEADERS_OR_SOURCE package or not.
# And not just "the" linux-headers (or source) package, because
# $LINUXHEADERS could be a specialised package (such as
# linux-headers-3.2.0-53-virtual) which installs include/trace as a
# symbolic link to the generic package (linux-headers-3.2.0-53).
# Note: We shut stderr up (2>&-) because the "normal" failure emits
# "dpkg-query: no path found matching pattern
#          *linux-headers*include/trace/hello.h*."
# PREEXTANT=`dpkg-query --search \
#           "$HEADERS_OR_SOURCE*include/trace/events/$TRACE_SYSTEM.h" 2>&-` \
#           || [ -z "" ]
[ ! -z "$PREEXTANT" ] && \
echo "Error: Your system name conflicts with an extant Linux" \
      "header. Choose a different name for your LTTng system." \
&& exit 1
echo "No system name conflict detected"
fi

echo "~~~~~Checking for lttn-probe-$TRACE_SYSTEM in" \
      "lttn-modules/probes/Makefile"
# Does the lttn-modules/probes/Makefile include
# lttn-probe-$TRACE_SYSTEM?
MAKEFILE_AMENDED=`cat $LTNGMODULES/probes/Makefile 2>&- | \
                  grep "lttn-probe-$TRACE_SYSTEM" ` || [ -z "" ]
if [ -z "$MAKEFILE_AMENDED" ]; then
  echo "Error: Your lttn-probe module won't be built by" \
        "lttn-modules."
  echo "Please amend $LTNGMODULES/probes/Makefile to include" \
        "lttn-probe-$TRACE_SYSTEM"
  exit 1
fi

```

```

echo "lttng-modules/probes/Makefile includes lttnng-probe-$TRACE_SYSTEM"

# Everything checks out: Go ahead!

echo "~~~~~Copying lttnng-probe-$TRACE_SYSTEM.c to lttng-modules/probes"
TARGET=$LTTNGMODULES/probes/lttnng-probe-$TRACE_SYSTEM.c
MSG="`${0##*/}` $*' requires write-access to $LTTNGMODULES/probes, etc."
gksudo --message "$MSG" -- \
    cp -b "lttnng-probe-$TRACE_SYSTEM.c" "$TARGET"
echo "lttnng-probe-$TRACE_SYSTEM.c copied to lttng-modules/probes"

# How to prepend a here-document to "./$TRACE_SYSTEM.h"?
# One possibility is to write the prepend lines to ./dummy, then
# cat the two:
#cat > ./dummy << ENDENDEND
#ENDENDEND
#cat ./dummy "./$TRACE_SYSTEM.h" > (destination file)
# Turns out we can do this instead:
echo "~~~~~Copying $TRACE_SYSTEM.h to $LINUXHEADERS_NAME"
# Normally you'd do just 'sudo cat ...' but the output redirect is done
# on the sudo output, not its argument command. So we sudo a sub-shell.
# Which means we must export our environment variables.
export TRACE_SYSTEM
export TARGET
TARGET=$LINUXHEADERS/include/trace/events/$TRACE_SYSTEM.h
# Backup any pre-extant target (cat's redirect can't do 'cp -b')
[ -f "$TARGET" ] && sudo -E -- mv --force "$TARGET" "$TARGET~"
# Now create the target
sudo -E -- sh -c 'cat > "$TARGET" << ENDENDEND
//These two #defines handle the subtle differences between
//the lttng-module and mainline versions of this file.
//For mainline:
#define _LTTNG_MAINLINE
#define _LTTNG_SEPARATOR ;

`cat "./$TRACE_SYSTEM.h"`
ENDENDEND'
echo "$TRACE_SYSTEM.h copied to Linux headers"

echo "~~~~~Linking lttng-modules...mainline/$TRACE_SYSTEM.h" \
    "to $LINUXHEADERS_NAME"
LINKTARGET=$LTTNGMODULES/instrumentation/events/mainline/$TRACE_SYSTEM.h
sudo -- ln -b --force --symbolic "$TARGET" "$LINKTARGET"
echo "lttng-modules...mainline $TRACE_SYSTEM.h linked to" \
    "Linux headers copy"

```

```

echo "~~~~~Copying $TRACE_SYSTEM.h to" \
    "lttng-modules/instrumentation/events/lttng-module"
TARGET=$LTTNGMODULES/instrumentation/events/lttng-module/$TRACE_SYSTEM.h
# Backup any pre-extant target (cat's redirect can't do 'cp -b')
[ -f "$TARGET" ] && sudo -E -- mv --force "$TARGET" "$TARGET~"
# Now create the target
sudo -E -- sh -c 'cat > "$TARGET" << ENDENDEND
//These two #defines handle the subtle differences between
//the lttng-module and mainline versions of this file.
//For lttng-module:
#define _LTTNG_MAINLINE
#define _LTTNG_SEPARATOR /* */

`cat "./$TRACE_SYSTEM.h"`
ENDENDEND'
echo "$TRACE_SYSTEM.h copied to lttng-modules...lttng-module"

if [ -z "$THE_MODULE" ]; then
    echo "~~~~~Skipping make of custom module"
else
    echo "~~~~~Creating Makefile for $THE_MODULE"
    export THE_MODULE
    TARGET=./Makefile
# Backup any pre-extant target
[ -f "$TARGET" ] && sudo -E -- mv --force "$TARGET" "$TARGET~"
# Now create the target
# Normally, the Makefile reads:
#     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) ...
# But the $ will be processed by bash, so we resolve the calls
# right in the here-document
    cat > "$TARGET" << ENDENDEND
obj-m += $THE_MODULE.o

all:
    make -C /lib/modules/`uname -r`/build M=`pwd` modules

clean:
    make -C /lib/modules/`uname -r`/build M=`pwd` clean
ENDENDEND

echo "~~~~~Making $THE_MODULE.ko"
make 2>&1 | tee ./make-$THE_MODULE.log
fi

```

```

echo "~~~~~Making lttng-probe-$TRACE_SYSTEM.ko"
cd $LTTNGMODULES
# sudo isn't strictly required, but we normally install to /usr/src...
sudo make 2>&1 | sudo tee make-lttng-probe-$TRACE_SYSTEM.log

echo "~~~~~Installing lttng-probe-$TRACE_SYSTEM.ko"
sudo make modules_install 2>&1 | \
    sudo tee install-lttng-probe-$TRACE_SYSTEM.log

echo "All done!"
exit $?

```

A.15 DRDC firewall palliation

The DRDC firewall relies on the McAfee Web Gateway (MAWG), which detects downloads of various types of executables (including shell scripts). From a Web browser, MAWG intercepts the request and presents the user with a page on which he can click to cancel the request or proceed with it. Next comes a request for file disposition (e.g. save to disk or open with the associated application), sometimes with an intermediate screen which presents a link to the requested file as cached by MAWG (because it took a while for MAWG to download and scan it).

This operational concept becomes troublesome when the download requests are meant to run unattended, such as when Synaptic attempts to fetch distribution packages from uncertified sites. The process is unaware of the hoops MAWG puts in the way and is unable to respond appropriately, the user being rarely apprised of the reason for the failure because it is an unexpected failure mode. MAWG interference manifests as updates or downloads that stall or fail more or less inexplicably.

When Synaptic reports the addresses of the files it has failed to fetch, you can copy them and go to the containing folder using your Web browser. Download the files manually to some convenient storage place. Once the download is done, you can click Synaptic's Apply button again: MAWG remembers you've cleared the file for download (for a few hours at least), so the second attempt should work correctly. You can then dispose of the first copy of the file you downloaded.

If you anticipate trouble of this sort or if you have a large number of packages to install or update, you can choose Synaptic's File: Generate package download script and save this script somewhere convenient. Open the script with a text editor such as gedit. It will contain a series of wget commands, the URL addresses of which give you the packages you need. Proceed to download each file manually.

Once you have all packages downloaded, copy them into /var/cache/apt/archives. This requires super-user privileges, so you can either open a super-user Nautilus window (`gksudo xdg-open /home/<username>/Downloads/<temp>` &) or copy them directly from the command line (`sudo cp /home/<username>/Downloads/<temp>/*.* /var/cache/apt/archives`). Substitute the appropriate values for <username> and <temp>.

Now go back to Synaptic and click the Reload button (your selection of installations and updates will remain unless you close Synaptic). This time, when you click Apply the Summary should state “0 B have to be downloaded”, because Synaptic will have found the files in its cache. At this point the installation or update can be completed without any connectivity issues.

A.15.1 Installing on a network-less system

A similar but more constraining situation is when one needs to install some packages on a system that is without an Internet connection (possibly because some installations are required before it can be connected). If the repository supplied with the Ubuntu installation CD does not suffice (or is out of date), here is what needs to be done.

A (partial) image of the repository or repositories must be built (on a CD-ROM or DVD-ROM, for instance). The contents of `http://archive.ubuntu.com/ubuntu/dists/precise/` (for an Ubuntu 12.04 repository) must be fetched. These describe the repository contents and do not take up too much total room. Other repositories may be needed, such as `archive.canonical.com`—the procedure is the same. It is possible to prune the tree of files to fetch if you know your target architecture (for instance, for a 64-bit target the `Contents-i386.gz` and `main/binary-i386/` can be omitted). Fetch also the packages of interest into the proper `pool/main` hierarchy. As long as you do not try to install anything else from that repository, a partial repository is just fine.

Finally, copy the repository to the target system (or mount the CD- or DVD-ROM) and add it to the `apt-get` list of sources. This is held in `/etc/apt/sources.list` (`sudo` privileges are required to edit it):

```
# /etc/apt/sources.list

deb cdrom:[Ubuntu 12.04.4 LTS _Precise Pangolin_ - Release amd64 (20140204)]/
      precise main
deb http://archive.ubuntu.com/ubuntu/ precise main
deb file:///repositories/archive.ubuntu.com/ubuntu/ precise main
```

A.15.2 Installing on a network-less virtual machine

On a virtual machine, disconnect the network card, add a second CD-ROM reader, boot with one loaded with the Ubuntu 12.04 `iso` image. Then insert the Virtual Box Guest Additions `iso` and install those in order to get access to the shared folder (albeit requiring `sudo` privileges because the user isn’t a member of the `vboxsf` group). Now add the correct subdirectory of the shared folder to the list of `apt` sources.

Annex B The `lttng` program

B.1 Synopsis

```
lttng [<OPTIONS>] <COMMAND> [<ARGUMENTS>]
```

B.2 Description

This description is based on the 2.3.0 release of the `lttng` command documentation. Some of the commands and options listed here may not be available when using the Ubuntu packaged version. The converse can also be true: some options listed in the Ubuntu packaged version may no longer exist (this often means they were not functional).

The `lttng` command line tool (from the `lttng-tools` package) is used to control both kernel and user-space tracing. Every interaction with the tracer should be done through this tool or through the `liblttng-ctl` API provided with the `git.lttng.org lttng-tools` tarball (or the older Ubuntu `liblttng-ctl0` package, if applicable).

LTTng uses a session daemon (`lttng-sessiond`, see Annex C), acting as a tracing registry, which permits you to interact with multiple tracers (kernel and user-space) inside the same container, a tracing session. Traces can be committed to storage by several consumer daemons (`lttng-ust`, see Annex I). Aggregating and reading those traces can be done using the `babeltrace` (see Annex F) trace converter.

LTTng introduces the notion of *tracing domains* which are essentially types of tracers (kernel or user-space for now). Version 2.4 (released on March 2nd, 2014) adds the `jul` domain, which captures `java.util.logging` events. In the future, other tracers could be added, such as for instance a hypervisor. For some commands, you'll need to specify to which domain the command applies (`-u` or `-k`). For instance, when enabling a kernel event you must specify the kernel domain to the command so LTTng can know which tracer this event is for.

In order to trace the kernel, the session daemon needs to be running as root. LTTng provides the use of a tracing group (default: `tracing`). Whoever is in that group interacts with the root session daemon using an unprivileged `lttng` client and thus can trace the kernel. Session daemons can co-exist, meaning that you can have a session daemon running as user `Alice` that can be used to trace her applications alongside with a root daemon or even a `Bob` daemon. The recommended installation automatically sets up the root session daemon as a service, spawned at system startup.

Most `lttng` commands will also automatically spawn a user-space session daemon if one isn't already running and the user isn't in the root daemon's tracing group. This spawning can be prevented by the `no-sessiond` option.

Every user-space application instrumented with `lttng-ust` (see Annex I) will automatically register with the session daemon of that user and the root session daemon, if they exist. This feature gives you the ability to list available traceable applications and tracepoints on a per-user basis. (See the `list` command, B.4.11)

Unless stated otherwise, names (sessions, channels, events, etc.), commands and options are case-sensitive and must match exactly. This is nearly always true of file system paths as well; however, the `http` protocol is a notable exception, as it is not case-sensitive as far as the domain name goes (the path—that is to say everything after the third slash—is case-sensitive, although servers are often configured to treat the entire address as case-insensitive).

B.3 Program options

This program follows the usual GNU command line syntax with long options starting with two dashes [48] (optional use of the '=' sign to separate long option names from their arguments is a GNU getopt extension). All options applicable to the `lttng` client itself must precede the `<COMMAND>`.

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` option pre-empts the `version` and `list-*` options if it appears first on the command line.

-V, --version

Shows the build version (e.g. ‘`lttng (LTTng Trace Control) 2.3.0 - Dominus Vobiscum`’) and then quits. When this option is specified, the remaining ones are ignored. This is the short form of the `version` command (B.4.16). The `version` option pre-empts the `help` and `list-*` options if it appears first on the command line.

--list-options

Shows a simple listing of the `lttng` options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` option pre-empts the `help`, `version`, and `list-commands` options if it appears first on the command line.

--list-commands

Shows a simple listing of the `lttng` commands and then quits. When this option is specified, the remaining ones are ignored. The `list-commands` option pre-empts the `help`, `version`, and `list-options` options if it appears first on the command line.

-v, --verbose

Increases verbosity. There are three debugging levels (`v`, `vv`, and `vvv`); you can also repeat `verbose` up to three times (repeating `v` or `verbose` more than three times is the same as omitting the option entirely [49]).

-q, --quiet

Suppresses all messages (even errors). The `quiet` option pre-empts the `verbose` one.

-g, --group <GROUP_NAME>

Allegedly sets the tracing group name for this command. Currently this option does absolutely nothing [50]. Which daemon (root or local) handles the command is determined as explained in Section 3.4. See the `group` option of `lttng-sessiond`, Section C.3, which serves a very different purpose.

-n, --no-sessiond

Prevents the spawning of a session daemon. This has an effect only if the relevant daemon is not currently running (in which case the command will often fail with exit value 19 (see Section B.5)). The `version` and `view` commands already do not spawn a daemon so they don't need this option. The only command that can meaningfully use this option is `set-session`, which normally spawns a daemon but does not need to do so.

--sessiond-path <PATH>

Sets the session daemon's full binary path. Normally, whenever `lttng` needs to spawn a session daemon it uses the installed one (`/usr/local/bin/lttng-sessiond`, found by the `whereis lttng-sessiond` command); this option provides an alternate path to the session daemon executable. It is mostly useful for developers who may want to run an experimental session daemon to replace the normal one.

B.4 Commands

The GNU `getopt` library is used to retrieve command line options and arguments. Because of this, the GNU syntax [48] is extended to allow long-named option arguments to be separated by the '=' sign in addition to the space. It also means command options can come before and after a command's argument. It is nevertheless recommended to stick to the command-options-argument sequence to dispel any possible ambiguity. This is reflected in the synopses given here (it also simplifies them a little).

B.4.1 add-context

Synopsis

```
lttng [<LTTNG_OPTIONS>] add-context (-t <TYPE>) + (-k|-u)  
[<COMMAND_OPTIONS>]
```

Description

Add context to a channel or all channels of a domain (kernel or userspace), which must be specified. You'll get an error if no session exists.

A context is basically extra information appended to the events of a channel¹². For instance, you could ask the tracer to add the PID (process identifier) information. You can also add performance monitoring unit (PMU) counters using the perf kernel API. For example, this command will add the context information 'prio' and a perf counter (hardware branch misses), to all kernel events in the trace data output:

```
# lttng add-context -k -t prio -t perf:branch-misses
```

If no channel is given, the context is added to all channels (and thus all events) of that domain. Otherwise the context will be added only to the channel indicated. If no channel is given and no channel yet exists, the default channel channel0 is created. If the session option is omitted, the (current) session name is taken from the .lttngrc file if it exists (it won't exist if no session has been created so far or if the last command was destroy; this will result in an error).

The command's help (`lttng add-context --help`) provides a detailed list of currently available contexts. Here is a sample of the 3.2.0-36-virtual kernel TYPES (the list of perf fields is incomplete):

hostname	perf:alignment-faults
ip	perf:branch-load-misses
nice	perf:branches
pid	perf:cache-misses
ppid	perf:context-switches
prio	perf:cpu-migrations
procname	perf:cycles
pthread_id	perf:faults
tid	perf:idle-cycles-backend
vpid	perf:page-fault
vppid	perf:stalled-cycles-frontend
vtid	perf:task-clock

¹² The Common Trace Format (CTF) also has an `event.context` structure (see Section F.2.2), but this is currently unused by lttng: you cannot add context at the event level, only at the channel level.

When adding `perf` fields, take care not to exceed the number of available generic registers for your machine. You can find that out with `dmesg | grep "generic register"` (if your machine's kernel ring buffer (which is what `dmesg` reads) has been emptied since boot time, try the command again shortly after a reboot). Also note that each time a `perf` field is added to a different channel, a separate instance is created. This means the same `perf` field will have different values in each channel; the values will increment at the same rate, however.

Currently the `add-context --help` command does not mention the domain specificity of each available context. By trial and error, you will find out that:

- ◆ the `ip` and `pthread_id` contexts are applicable only to the user-space domain,
- ◆ the `procname`, `vpid` and `vtid` contexts are applicable to both the user-space and kernel domains, and
- ◆ all other contexts (including the `perf` fields) are applicable only to the kernel domain.

Each context can only be added once to any given channel. Repeating an `add-context` command will yield the same confirmation message but the context will appear just once in the events. It is possible to add context to some specific channels and then add the same context to all channels; this produces no warning or error messages. Adding context to all channels only affects the channels that exist at that point in time: if you create additional channels afterward, they won't have any context added.

The `list <session>` command does not currently list the contexts added to each channel [51].

There is currently no means of removing added context from a channel. Since there is neither any means of removing a channel once created, the only option is to scrap the session and start again. In user-space, you could also disable the channel, enable a properly configured new one, and assign the old channel's events to the new channel. You can't do that in the kernel domain, because the kernel events can only be assigned once.

Context appears in the `babeltrace` output in the `stream.event.context` structure, one field per context (see Section F.2.2). For example, here is a `babeltrace` output fragment for a trace where the `ip` context had been added to some user-space channel:

```
[...] stream.event.header = { [...] },  
stream.event.context = { ip = 0x400E31 }, event.fields = [...]
```

The context fields appear in the `babeltrace` output in the order in which they were added to the channel. For example, if you use the '`lttng add-context -u -t ip -t pthread_id`' command, the `stream.event.context` structure will list the `ip` field first, followed by the `pthread_id` field. If you use the '`lttng add-context -u -t pthread_id -t ip`' command instead, the `stream.event.context` structure will list the fields in the reverse order.

The names of `perf` fields contain colons (‘:’) and sometimes also hyphens (‘-’, e.g. `perf:context-switches`). They appear in the trace with these characters replaced by underscores (‘_’, e.g. `perf_context_switches`). This shows in the resulting `babeltrace` output, and will also be important for the event filter option once it becomes available to the kernel domain.

The context fields are accessible to any filter added to an event of the channel, *whether or not add-context has been used*. The `add-context` command only serves to make the chosen context fields explicit in the trace. In other words, you need not `add-context` to a channel in order to use the context in a filter. See the `filter` option of the `enable-event` command for details (Section B.4.10).

Command options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` command option pre-empts the `list-options` command option if it appears first on the command line.

--list-options

Shows a simple listing of the `lttng add-context` options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` command option pre-empts the `help` command option if it appears first on the command line.

-k, --kernel

Applies the context to the kernel tracer (there is no default). A domain must be specified.

-u, --userspace

Applies the context to the user-space tracer (there is no default). A domain must be specified. The `kernel` option pre-empts this option.

-s, --session <SESSION_NAME>

Applies the context to the session `<SESSION_NAME>`. Defaults to the current session. If there is no current session, an error occurs.

-c, --channel <CHANNEL_NAME>

Applies the context to the channel <CHANNEL_NAME>. The channel must already exist. If unspecified, the context is applied to all of the session's currently extant channels. Should there exist no channels, the default channel “channel0” is created (quietly: no message announces this) and the context added to it.

-t, --type <TYPE>

Specifies a context type. At least one type must be specified. You can repeat this option on the command line to specify multiple contexts at once.

B.4.2 calibrate

Synopsis

```
lttng [<LTTNG_OPTIONS>] calibrate (-k|-u) [<COMMAND_OPTIONS>]
```

Description

This command does not work yet, and getting it fixed is considered low priority [52]. The intent is for it to trigger a calibration event that, if instrumented and captured by a tracing session, can be used to quantify the combined average overhead of the LTTng tracer and the instrumentation mechanism used. This overhead will be calibrated in terms of time or using any of the PMU performance counters (`perf`) available on the system.

As of this writing, the only calibration events implemented are a kernel function and a user-space event (which, when triggered, is emitted by each registered application). A variety of other calibration events are considered for future implementation (e.g. tracepoints, dynamic probes, system calls).

The `lttng calibrate` command is context-less: the event(s) it triggers may be captured by any number of sessions. Currently, the calibration events are not implemented correctly and cannot be used (babeltrace will be unable to read the trace produced).

Command options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` command option pre-empts the `list-options` command option if it appears first on the command line.

--list-options

Shows a simple listing of the `lttng calibrate` options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` command option pre-empts the `help` command option if it appears first on the command line.

-k, --kernel

Applies the calibration to the kernel tracer (there is no default). A domain must be specified.

-u, --userspace

Applies the calibration to the user-space tracer (there is no default). A domain must be specified. The `kernel` option pre-empts this option.

--function

Specifies that the calibration should use the dynamic function entry/return probe (this is the default and currently the only choice available).

B.4.3 create

Synopsis

```
lttng [<LTTNG_OPTIONS>] create [<SESSION_NAME>]  
[<COMMAND_OPTIONS>]
```

Description

Create a tracing session (see 2.2 for a summary of what sessions are).

Upon its creation, the tracing session is given the specified `<SESSION_NAME>`. If no `<SESSION_NAME>` is specified, it defaults to ‘auto-<yyyymmdd>-<hhmmss>’, using its creation timestamp. The `<SESSION_NAME>` can be any legal file name except ‘auto’ or anything beginning with ‘auto-’, and can in theory be up to 245 characters in length, although names should be considerably shorter.

You can be perverse and use the slash (‘/’) in the `<SESSION_NAME>`; this will merely introduce unexpected subfolders in the trace path but won’t confuse babeltrace. It may, on the other hand, merge traces: for example, if you ran two traces called respectively `auto/one` and `auto/two`, `babeltrace ~/lttng-traces/auto` would merge the two traces. You may even use ‘`./`’ and/or ‘`../`’ in the `<SESSION_NAME>`, but `lttng` will reject the resulting trace folder path when you start the trace. Starting with version 2.4, no ‘/’ may be used in a session name.

You use the `start` and `stop` commands (see B.4.13 and B.4.15) to make the session respectively active or inactive. Eventually, the tracing session is destroyed (see B.4.4), closing its data files (the trace) so they can henceforth be safely manipulated (copied, archived, etc.).

The trace folder's name (when stored locally) is the tracing session's name suffixed by a timestamp of the form '`-<yyyymmdd>-<hhmmss>`' (except for the default '`auto-<yyyymmdd>-<hhmmss>`', which is not suffixed a second time). This timestamp suffix allows users to re-use tracing session names without fear of overwriting previously saved trace data. The trace root folder is written in `$HOME/lttng-traces` unless the `output` option is specified.

Alternately, the trace data may be streamed over the network to a remote destination using the `set-url`, `ctrl-url`, and `data-url` options. There must be a running remote `lttng-relayd` daemon (or compatible equivalent) for network streaming (see Annex D). In older releases, the trace data's output destination could be changed after session creation but before trace start by the `enable-consumer` command.

In order to be able to manage multiple sessions without having to specify the session every time an `lttng` command is issued (although you can do this if you want), each daemon keeps track of a “current” session, and the `set-session` command is used to switch between them. This is done by the `.lttngrc` file in the `$HOME` folder of the user (or of root's). Note that when the current session is destroyed, the `.lttngrc` file is deleted because there is no longer a current session. Creating a session makes it the current one.

Examples

Use TCP and default ports for the given destination:

```
# lttng create -U net://192.168.1.42
```

Use TCP, default ports and IPv6:

```
# lttng create -U net6://[fe80::f66d:4ff:fe53:d220]
```

Create session ‘`s1`’ and tell its consumers to write to `myhost.com`, using port 3229 for control (the data stream will use default port 5343):

```
# lttng create s1 -U net://myhost.com:3229
```

Command options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` command option pre-empts the `list-options` command option if it appears first on the command line.

--list-options

Shows a simple listing of the `lttng create` options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` command option pre-empts the `help` command option if it appears first on the command line.

--no-output

Prevents the trace from being output at all. This is one way to put the trace in snapshot mode; see the `snapshot` command (Section B.4.13) for the details. No `lttng-consumerd` daemons will be spawned. The `ctrl-url` and `data-url` options pre-empt the `output` option, which pre-empts the `set-url` option, which pre-empts the `no-output` option.

-U, --set-url <URL>

Sets the URL (Uniform Resource Locator) for the trace data stream destination. The `ctrl-url` and `data-url` options pre-empt the `output` option, which pre-empts the `set-url` option, which pre-empts the `no-output` option. This will set both the data and control stream URLs for network transmission. The supported URL formats are detailed in the sub-section immediately following the command options sub-section. This option, unlike the now-obsolete `enable-consumer -U` command option, recognises a local file path: prefixing with `file://` is not necessary. **Note:** Before the 25 March 2013 23d14df commit, `create` recognises ‘`set-uri`’ instead of ‘`set-url`’.

Example:

```
$ sudo -H lttng create -U net://131.132.32.66
$ sudo -H lttng enable-event sched_switch -k
```

-o, --output <PATH>

Specifies the output path of the session’s trace (the path to the *session folder*). The `ctrl-url` and `data-url` options pre-empt the `output` option, which pre-empts the `set-url` option, which pre-empts the `no-output` option. By default, the `<PATH>` is `$HOME/lttng-traces/<SESSION_NAME>-<date>-<time>`. A `<PATH>` of length 3474 characters or less should be safe (excluding the trailing ‘/’ and null).

-C, --ctrl-url <CTRL_URL>

Sets the control stream path to <CTRL_URL>. The `ctrl-url` and `data-url` options pre-empt the `output` option, which pre-empts the `set-url` option, which pre-empts the `no-output` option. If <CTRL_URL> is not supplied or is a `file:`, an error occurs. If you do not also supply a `data-url` option, an error occurs. The supported URL formats are detailed in the sub-section immediately following the command options sub-section. Using `tcp:` or `tcp6:`, if you set the `ctrl-url` and `data-url` options to the same port, the command fails with a laconic “Invalid parameter” error message. **Note:** Before the 25 March 2013 23d14df commit, `create` recognises ‘`ctrl-uri`’ instead of ‘`ctrl-url`’.

In brief, there are three ways of creating a session that is streamed to a remote destination, and two of creating a session that is stored locally:

```
$ lttn create <remote> --set-url net://<address>:<ctrl>:<data>
$ lttn create <remote> --ctrl-url tcp://<address>:<ctrl> \
                         --data-url tcp://<address>:<data>
$ lttn create <remote> --ctrl-url net://<address>:<ctrl>:<data> \
                         --data-url net://<address>:<ctrl>:<data>
$ lttn create <local>  --output <path>
$ lttn create <local>  --set-url file://<path>
```

-D, --data-url <DATA_URL>

Sets the data stream path to <DATA_URL>. The `ctrl-url` and `data-url` options pre-empt the `output` option, which pre-empts the `set-url` option, which pre-empts the `no-output` option. If <DATA_URL> is not supplied, an error occurs. If you do not also supply a `ctrl-url` option, an error occurs. When `ctrl-url` is a `net:` or a `net6:`, and `data-url` is not *strictly* identical (canonical equivalence does not suffice: e.g. ‘`-C net6://[:8384:204d] -D net6://[:8384:204D]`’ is a mismatch), an error occurs. The supported URL formats are detailed in the sub-section immediately following the command options sub-section. **Note:** Before the 25 March 2013 23d14df commit, `create` recognises ‘`data-uri`’ instead of ‘`data-url`’.

--no-consumer

This option is obsolete since 2013 February 18; it is recognised by `lttn` but no longer acted upon (a warning is issued). Prevents the consumer daemons from spawning for this session. Use the (now also obsolete) `enable-consumer` command (see B.4.9) to spawn them before starting the session. The `ctrl-url` and `data-url` options used to pre-empt `no-consumer`, which pre-empted the `output` option, etc.

--disable-consumer

This option is obsolete since 2013 February 18; it is recognised by lttng but no longer acted upon (a warning is issued). Disables the (already spawned) consumer daemons for this session. Use the (now also obsolete) enable-consumer command (see B.4.9) to re-enable them before starting the session.

--snapshot

This option was introduced by LTTng 2.3.0-rc1 (July 17, 2013). Subsumes the no-output option and sets every channel to come in overwrite mode with the output option set to mmap (splice is not supported). See the snapshot command, Section B.4.13.

URL format

```
(file|net|net6|tcp|tcp6):// [<HOST>|<IP>] [:<PORT1>[:<PORT2>]]  
[/<TRACE_PATH>]
```

The file protocol expects a local file system full path. You may not start from the ~ folder; all parts of the <TRACE_PATH> except its leaf folder must already exist and be writable.

The command's argument and its ctrl-url and set-url options (but not the data-url option) also accept the following format (it is automatically prefixed with file://):

```
/TRACE_PATH
```

The net protocol will use the default network transport layer which is TCP for both the control (<PORT1>) and data (<PORT2>) stream ports. The default ports are respectively 5342 and 5343. The udp (User Datagram Protocol) protocol may become available for the data stream in the near future, as should serial interfaces. UDP will never be used for the control stream because the latter requires reliable, ordered, error-checked delivery. **Notes:** Older help and man pages mention the tcp4 protocol, but this was never recognised; also, the net6 protocol is not yet supported.

An IPv6 address *must* be enclosed in brackets '[]' [53]. An IPv6 address can drop leading zeroes (e.g. ':4ff:' instead of ':04ff:'), replace a single group of zeroes (of any length) with '::', and write the last two groups in IPv4 notation [54]. The following IPv6 addresses are thus all equivalent:

```
[fe80:0000:0000:0000:f66d:04ff:fe53:d220]  
[fe80:0000:0000:0000:f66d:4ff:fe53:d220]  
[fe80:0000:0000:0000:f66d:04ff:254.83.210.32]  
[fe80:0000:0000:0000:f66d:4ff:254.83.210.32]  
[fe80::f66d:04ff:fe53:d220]  
[fe80::f66d:4ff:fe53:d220]  
[fe80::f66d:04ff:254.83.210.32]  
[fe80::f66d:4ff:254.83.210.32]
```

Examples

Using TCP and default ports:

```
$ lttnng create <session> -U net://192.168.1.42
```

The IP address is that of the destination machine, where the traces will be streamed and an `lttnng-relayd` is presumed to be listening.

B.4.4 destroy

Synopsis

```
lttnng [<LTTNG_OPTIONS>] destroy [<SESSION_NAME>]  
[<COMMAND_OPTIONS>]
```

Description

Tear down a tracing session. If the session is currently active, the command first stops it.

This command frees session-related memory held by the session daemon and tracer(s). This is irreversible. If `<SESSION_NAME>` is omitted, the current session name is taken from the `.lttnngrc` file if it exists (it won't exist if no session has been created so far or if the last command was `destroy`; this will result in an error). When the current session is destroyed (implicitly or explicitly), the `.lttnngrc` file is deleted and you will therefore have to specify which is the new current session using the `set-session` or `create` commands.

Options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` command option pre-empts the `list-options` command option if it appears first on the command line.

--list-options

Shows a simple listing of the `lttnng destroy` options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` command option pre-empts the `help` command option if it appears first on the command line.

-a, --all

Destroys all of the user's sessions. Note that (see Section 2.2.1) for a user belonging to the tracing group, this option will destroy only his own sessions, whereas for the root user (`sudo lttng destroy --all`) this option will destroy all sessions managed by the root session daemon, regardless of which user created them.

B.4.5 disable-channel

Synopsis

```
lttng [<LTTNG_OPTIONS>] disable-channel \
       [<CHANNEL_NAME>[,<CHANNEL_NAME2>, ...] (-k|-u)] [<COMMAND_OPTIONS>]
```

Description

Disable tracing channel(s).

Disabling a channel suspends tracing the events from that channel. You can enable it back by calling `lttng enable-channel <CHANNEL_NAME>`; the flow of events then resumes. If the `session` option is omitted, the current session name is taken from the `.lttngrc` file if it exists (it won't exist if no session has been created so far or if the last command was `destroy`; this will result in an error). At least one `<CHANNEL_NAME>` must be specified (unless the `help` or `list-options` option is specified).

Options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` command option pre-empts the `list-options` command option if it appears first on the command line.

--list-options

Shows a simple listing of the `lttng disable-channel` options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` command option pre-empts the `help` command option if it appears first on the command line.

-s, --session <SESSION_NAME>

Applies the command to the `<SESSION_NAME>` session. Defaults to the current session. If there is no current session, an error occurs.

-k, --kernel

Applies the command to the kernel tracer (there is no default). A domain must be specified. Channels are local to each domain.

-u, --userspace

Applies the command to the user-space tracer (there is no default). A domain must be specified. The `kernel` option pre-empts this option.

B.4.6 disable-consumer

Synopsis

```
lttng [<LTTNG_OPTIONS>] disable-consumer [-k|-u]
[<COMMAND_OPTIONS>]
```

Description

This command is obsolete since 2013 February 18; it is recognised by `lttng` but no longer acted upon (a warning is issued). Disable the consumers of a tracing session. This call must be done before tracing has started.

Options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` command option pre-empts the `list-options` command option if it appears first on the command line.

--list-options

Shows a simple listing of the `lttng disable-consumer` options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` command option pre-empts the `help` command option if it appears first on the command line.

-s, --session <SESSION_NAME>

Applies the command to the `<SESSION_NAME>` session. Defaults to the current session. If there is no current session, an error occurs. Since all the sessions handled by a given session daemon are serviced by at most three consumer daemons (one for the kernel domain, two for user-space), this command really just tells the consumer daemons to suspend all activity on behalf of that session. The other sessions will continue to be serviced.

-k, --kernel

Applies the command to the kernel tracer (there is no default). A domain must be specified. Each domain is serviced by separate consumer daemons (shared between sessions).

-u, --userspace

Applies the command to the user-space tracer (there is no default). A domain must be specified. The `kernel` option pre-empts this option.

B.4.7 disable-event

Synopsis

```
lttng [<LTTNG_OPTIONS>] disable-event \
[<EVENT_NAME>[,<EVENT_NAME2>, ...] \
(-k|-u) [<COMMAND_OPTIONS>]
```

Description

Disable tracing event(s). An event, once disabled, can be re-enabled by calling `lttng enable-event <EVENT_NAME>`. If the `session` option is omitted, the current session name is taken from the `.lttngrc` file if it exists (it won't exist if no session has been created so far or if the last command was `destroy`; this will result in an error).

The command disables all event entries that match the `<EVENT_NAME>`, regardless of their `loglevel` and `filter` attributes. It is currently not possible to disable a specific `enable-event` entry when several entries share the same name part. See the `enable-event` command (Section B.4.10) for a discussion of LTTng's event enabling scheme. The `<EVENT_NAME>` must match an `enable-event` entry exactly, including any wildcard character (user-space only), or else the command fails and an error message is issued. When using a wildcard, be sure to quote the `<EVENT_NAME>` to avoid shell command line expansion: for instance, if there is a file called `babel` in your current working folder, the command `lttng disable-event bab* -u` will try to disable the `babel` event, not `bab*` as expected.

The command also fails if the channel does not exist. Unlike the `enable-event` command, if the `channel` option is omitted, a default channel named ‘`channel0`’ is *not* created if the domain is ‘`virgin`’ [55].

Examples

The following examples illustrate how the `enable-event` command works.

First example:

```
$ lttng list session
[...]
Events:
* (type: tracepoint) [enabled]
sw* (type: tracepoint) [enabled]
$ lttng disable-event "sw*" -u
UST event sw* disabled in channel channel0 for session session
$ lttng list session
[...]
Events:
* (type: tracepoint) [enabled]
sw* (type: tracepoint) [disabled]
```

Second example:

```
$ lttng list session
[...]
Events:
* (type: tracepoint) [enabled]
sw* (type: tracepoint) [enabled]
$ lttng disable-event "s*" -u
Error: Event s*: UST event not found (channel channel0, session session)
Warning: Some command(s) went wrong
```

Third example:

```
$ lttng list session
[...]
Events:
* (type: tracepoint) [enabled]
sw* (type: tracepoint) [enabled]
$ lttng disable-event "*" -u
UST event * disabled in channel channel0 for session session
$ lttng list session
[...]
Events:
* (type: tracepoint) [disabled]
sw* (type: tracepoint) [enabled]
```

Fourth example:

```
$ lttnng list session
[...]
    Events:
        * (type: tracepoint) [enabled]
        sw* (type: tracepoint) [enabled]
$ lttnng disable-event -a -u
All UST events are disabled in channel channel0
$ lttnng list session
[...]
    Events:
        * (type: tracepoint) [disabled]
        sw* (type: tracepoint) [disabled]
```

Fifth example:

```
$ lttnng list session
[...]
    Events:
        abcd (type: tracepoint) [enabled]
        abcd (loglevel: TRACE_WARNING (4)) (type: tracepoint) [enabled]
        sw* (type: tracepoint) [enabled]
$ lttnng disable-event abcd -u
UST event abcd disabled in channel channel0 for session session
$ lttnng list session
[...]
    Events:
        abcd (type: tracepoint) [disabled]
        abcd (loglevel: TRACE_WARNING (4)) (type: tracepoint) [disabled]
        sw* (type: tracepoint) [enabled]
```

Options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` command option pre-empts the `list-options` command option if it appears first on the command line.

--list-options

Shows a simple listing of the `lttnng disable-event` options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` command option pre-empts the `help` command option if it appears first on the command line.

-s, --session <SESSION_NAME>

Applies the command to the <SESSION_NAME> session. Defaults to the current session. If there is no current session, an error occurs.

-c, --channel <NAME>

Applies the command to channel <NAME>. Can be omitted only while the session has a single, default channel, “channel0” (the command will not create the default channel [55]). If “channel0” was explicitly enabled, it is *not* considered a default channel.

-a, --all-events

Disables all events of the channel. If there are syscall events assigned to the channel (see the enable-event command, Section B.4.10), they are not disabled and an error message stating this is issued. This is normal with the current LTTng version: system call tracing can only be disabled by disabling the *channel* (the error message recommends destroying the session, but this is unnecessary).

This does *not* disable “*” but rather every known event of the session (compare the third and fourth examples, above).

-k, --kernel

Applies the command to the kernel tracer (there is no default). A domain must be specified.

-u, --userspace

Applies the command to the user-space tracer (there is no default). A domain must be specified. The `kernel` option pre-empts this option.

B.4.8 enable-channel

Synopsis

```
lttng [<LTTNG_OPTIONS>] enable-channel \
[<CHANNEL_NAME>[,<CHANNEL_NAME2>,...] \
(-k|-u) [<COMMAND_OPTIONS>] [<CHANNEL_OPTIONS>]
```

Description

Enable one or more tracing channels (<CHANNEL_NAME>, <CHANNEL_NAME2>, etc.) and optionally set some of their attributes. At least one <CHANNEL_NAME> must be specified (unless the help or list-options options are specified). Channel names are typically limited to 254 characters in length (this global limit is specified by linux-headers/include/linux/limits.h) and are arbitrary; however, the command treats the comma as a channel name separator, so channel names cannot include commas. Channel names may include spaces, but these will need to be escaped in some contexts (e.g. ‘lttng disable-channel -k my\ kernel\ channel’). Channel names should not include character sequences that are meaningful to the file system (this is not currently checked by LTTng but may be in a future release) as this will typically prevent the trace from being written to storage (no error is issued but no trace file is created) [56].

If the session option is omitted, the current session name is taken from the .lttngrc file if it exists (it won’t exist if no session has been created so far or if the last command was destroy; this will result in an error). Any <CHANNEL_OPTIONS> specified apply to all of the <CHANNEL_NAME>s.

If you enable an already-enabled channel (or disable an already-disabled one), LTTng complains that the “Channel already exists” and ignores any channel options you may have set. Note that when you create an event (see enable-event, B.4.10) its channel is created and enabled, so if you want settings other than the defaults, you *must* enable the channel *before* creating its events. Once created, a channel’s attributes may not be changed. Furthermore, once a session has started, channels may no longer be added, even if the session is paused. This restriction may eventually be lifted.

The individual enabled/disabled settings of the events in a channel are persistent; the enabled/disabled setting of each channel acts simply as a master switch. In other words, an event is enabled if it is set to “enabled” *and* its channel is also set to “enabled”; any other combination of settings yields a disabled event.

Once you have created a channel explicitly within a domain, the channel option of the enable-event command must be specified or else you get a “Non-default channel exists within session: channel name needs to be specified” error.

Options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The help command option pre-empts the list-options command option if it appears first on the command line.

--list-options

Shows a simple listing of the `lttng enable-channel` options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` command option pre-empts the `help` command option if it appears first on the command line.

-s, --session <SESSION_NAME>

Applies the command to the `<SESSION_NAME>` session. Defaults to the current session. If there is no current session, an error occurs.

-k, --kernel

Applies the command to the kernel tracer (there is no default). A domain must be specified.

-u, --userspace

Applies the command to the user-space tracer (there is no default). A domain must be specified. The `kernel` option pre-empts this option.

Channel options

--discard

Sets the channel in discard mode: if the sub-buffers are full, new events are discarded. This is the default. The opposite setting is the `overwrite` option. If you put both options on the command line, the first one is ignored.

--overwrite

Sets the channel in flight recorder mode: if the sub-buffers are full, the oldest events are overwritten. The opposite setting is the `discard` option. If you put both options on the command line, the first one is ignored.

--subbuf-size <SIZE>

Sets the sub-buffer size. `<SIZE>` is in bytes (default: 4096 for user-space per PID, 131 072 for user-space per UID, 262 144 for kernel). The suffixes K, M, or G may be used to specify respectively kibibytes, mebibytes, or gibibytes (the suffix k is also taken to mean kibi). The `<SIZE>` will be rounded up to a power of 2 (a warning to this effect is issued). For instance, setting `subbuf-size` to `262K` requests a sub-buffer size of $262 \times 1024 = 268,288$ bytes which is rounded up to 512 Kib = 524,288 bytes. The minimum sub-buffer size (for both domains and either buffering scheme) is given by the system's `PAGESIZE`. You can obtain this by issuing the '`getconf PAGESIZE`' command. A `<SIZE>` prefixed with `0x` is understood to be expressed in hexadecimal, while a `<SIZE>` prefixed with just a leading zero is understood to be expressed in octal. `<SIZE>` can be up to 2^{63} (9,223,372,036,854,775,808, hexadecimal 8000 0000 0000 0000, 8 exabytes) inclusive.

Currently, `lttng` rejects hexadecimal `<SIZE>` values that contain the hexadecimal digits A through F, and does not recognise the `0x` hexadecimal prefix (which is standard C) [57]. There is also a rounding problem [58]: if you specify a $2^{64} > <SIZE> > 2^{63}$, `lttng` will “round it up” to 1 instead of rejecting it. This also applies to `tracefile-size` and `num-subbuf`. The latter bug is fixed in version 2.4.

--num-subbuf <NUM>

Sets the number of sub-buffers (default: 4). `<NUM>` will be rounded up to a power of 2 (a warning to this effect is issued). `<NUM>` must be 1 or greater. `<NUM>` can be expressed using decimal, octal or hexadecimal notation (the `0x` hexadecimal prefix is correctly handled). In overwrite mode, can be as large as 2^{63} (9,223,372,036,854,775,808, hexadecimal 8000 0000 0000 0000, 8 exbi) inclusive (for a 64-bit architecture). In discard mode, can be as large as 2^{32} (4,294,967,296, hexadecimal 1 0000 0000, 4 gibi). Of course, LTTng will run out of buffer memory long before this number of sub-buffers can be reached. Currently, specifying too large a `<NUM>` for the machine's physical memory will make the channel unusable but no error will be reported [59].

Each channel requires a little more than $<\text{MULTIPLICITY}> \times <\text{NUM}> \times <\text{SIZE}> \times <\text{NCPU}>$ bytes of memory, where `<NCPU>` is the number of CPUs being traced and `<MULTIPLICITY>` is the channel's multiplicity. In per-user-ID mode (see the `buffers-uid` option, below), this will be the number of active user IDs; in per-process-ID mode (see the `buffers-pid` option, below), it will be the number of active processes. In overwrite mode, each buffer uses an extra sub-buffer¹³, so the formula becomes $<\text{MULTIPLICITY}> \times (<\text{NUM}> + 1) \times <\text{SIZE}> \times <\text{NCPU}>$ bytes.

¹³ This extra sub-buffer serves to allow the reader to finish reading a sub-buffer while writers are overwriting it. This prevents the reader from blocking the writers, and dispenses the writers from interrupting the reader. You can think of the extra sub-buffer as a sort of “passing lane” for the writers.

-C, --tracefile-size <SIZE>

Sets the maximum size of each stored trace file (local or remote) within a stream, in bytes. Default is zero, which means no limitation. When set, each trace file is chopped into sequentially numbered chunks of at most <SIZE> bytes each (e.g. “channel0_0” becomes “channel0_0_0”, “channel0_0_1”, etc.). <SIZE> must be at least as large as the sub-buffer size. The option is most useful if the `tracefile-count` option is also set. For the full range of values <SIZE> can take, see the `subbuf-size` option (same syntax and limits).

Currently, `lttng list <session>` won’t give the `tracefile-size` and `tracefile-count` of the channels [60].

-W, --tracefile-count <NUM>

Sets the maximum number of stored trace file chunks (local or remote). Default is zero, which means no limitation. When set, once the number of trace file chunks reaches <NUM> the first chunks are re-used (overwritten), creating an on-disc circular buffer of sorts. Note that there is no relation between this command option and the channel’s `discard/overwrite` mode setting. The option will be rejected if it is not accompanied by a `tracefile-size` option (it is meaningless without it). <NUM> can be expressed using decimal, octal or hexadecimal notation (the `0x` hexadecimal prefix is correctly handled) and can be as large as $2^{31}+2^{28}-1$ (2,415,919,103, hexadecimal `8FFF FFFF`, 2.25 gibi minus one) inclusive.

Currently, `lttng list <session>` won’t give the `tracefile-size` and `tracefile-count` of the channels [60].

--switch-timer <USEC>

Sets the switch sub-buffer timer interval in `usec` (default: 0). Zero means “never force switch”. At the end of each switch interval, the current sub-buffer is padded out so that the next sub-buffer becomes the current one; this also makes the ex-current sub-buffer available for consumption by the channel reader (which matters only for live tracing). The number of sub-buffers (the `num-subbuf` option) should be greater than 1 for this to be meaningful. For the full range of values <USEC> can take, see the `tracefile-count` option (same syntax and limits).

--read-timer <USEC>

Sets the read timer interval in `usec` (default: 0 for user-space, 200,000 for kernel). At the end of each read interval, pending reader threads are woken up. For the full range of values <USEC> can take, see the `tracefile-count` option (same syntax and limits). Zero means no timer.

--output <TYPE>

Sets the channel output type. <TYPE> may be `mmap` (the user-space default) or `splice` (the kernel default). The <TYPE> defines the data transport service used to move trace data. The `mmap` transport service uses memory-mapped files while the `splice` transport service uses pipe splicing (sometimes called zero-copying). Splicing allows data to be moved around within the kernel without needing to actually copy it to an intermediate user-space buffer, avoiding expensive context switches from the kernel to user-space and back. *Splicing is only available for the kernel domain*, meaning user-space channels must use `mmap` but kernel channels may use either `splice` or `mmap`.

--buffers-global

Tells the channel to use a single set of buffers. This buffering scheme applies to the kernel domain only and is currently its only option.

--buffers-pid

Tells the channel to use per-PID (per-process-ID) buffers. This buffering scheme applies to user-space only. Although the setting is declared at the channel level, the buffering scheme applies to the entire domain (all other channels of the domain —this constraint may be relaxed eventually). Even if the session has not been started yet, the setting may not be changed. Any later `enable-channel` commands using a different buffering scheme will be rejected. The default is per-UID (earlier releases had a default of per-PID).

Compared with the `buffers-uid` option, per-PID buffers consume more memory and have more performance overhead, particularly when the trace is dealing with numerous short-lived processes. When a trace provider registers itself with the session manager, it blocks until registration is complete (in particular until the buffer is set up): this overhead may be deemed unacceptable with short-lived processes that ought to be fast. The trace is more robust, with fewer events lost when a traced process dies suddenly (in particular, the trace can only lose events issuing from the dying process). Individual process trace files become available as soon as each process concludes, facilitating their access (this will become moot once live trace viewing is implemented).

--buffers-uid

Tells the channel to use per-UID (per-user-ID) buffers. This buffering scheme applies to user-space only. Although the setting is declared at the channel level, the buffering scheme applies to the entire domain (all other channels of the domain —this constraint may be relaxed eventually). Even if the session has not been started yet, the setting may not be changed. Any later `enable-channel` commands using a different buffering scheme will be rejected. The default is per-UID (earlier releases had a default of per-PID).

The user ID used to sort the event streams is the real user ID (RUID). Compared with the `buffers-pid` option, per-UID buffers consume less memory and have less performance overhead, particularly when the trace is dealing with numerous short-lived processes. All the processes belonging to a given user ID use the same buffer. The trace is less robust: if a process dies suddenly it may create a “hole” in an event sub-buffer, causing the loss of a number of unrelated events (events issued by other processes) as well as forcing the consumer to wait for the loss to be confirmed (this occurs when a tracer tries to re-use the sub-buffer) [61].

Also note that per-UID traces lack the process ID contextual attributes present in per-PID traces (see the `PROCNAME` and `VPID` entries in Section F.2.2), unless these are added to each channel by the `add-context` command (see Section B.4.1) or supplied by special-purpose event payloads.

Currently, the `buffers-pid` option must be used with *all* user-space `enable-channel` commands of the session if it is used at all.

B.4.9 enable-consumer

Synopsis

```
lttng [<LTTNG_OPTIONS>] enable-consumer [<URL>] [-k|-u]
[<COMMAND_OPTIONS>]
```

Description

This command is obsolete since 2013 February 18; it is recognised by `lttng` but no longer acted upon (a warning is issued). Enable consumers for the tracing session and domain.

By default, every tracing session domain has one or two consumers (the user-space domain may have 32- and 64-bit consumers) attached to it which use the local file system to output the trace (by default to `$HOME/lttng-traces`; see the `output` option of `create`, B.4.3). This command allows the user to specify another `<URL>` for the consumer output. You may reissue the command repeatedly, the newer settings overriding the older ones, but once the session is started the settings will remain for the session’s lifetime and can no longer be changed, even if it is paused. It is possible to assign different consumer configurations for each domain (e.g. two different remote storage devices for the kernel and user-space traces). If no domain is specified, the command is applied to both domains.

The `enable-consumer` feature supports both local and network transport. For network transmission, you must have `lttng-relayd` (or any other process that can understand the LTTng streaming protocol) running on the target (typically as a daemon). Depending on the specific sequence of session commands used, the default folder may already have been created by the time you specify a new consumer target, local or remote; if the consumers are redirected elsewhere, the local folder will simply remain empty.

Specifying a local file system target, using either the command's argument or any one of the three options `set-uri`, `ctrl-uri`, or `data-uri`, means the trace will be stored there. The target can be of the `/<path>` or `file:///<path>` form except for the option `set-uri`, which requires the `file:///<path>` form. The command creates the folder (and its domain subfolders) immediately.

Specifying a remote file system target currently means using either of the `net:` or `tcp:` protocols (see the *URL format* sub-section, below). Besides a host name or IP address (v4 or v6), these protocols use separate ports for the control and data streams (by default 5342 and 5343, respectively). Again, this can be done using either the command's argument or any one of the three options `set-uri`, `ctrl-uri`, or `data-uri`. The remote folders are created when the trace is started.

When using the `set-uri`, `ctrl-uri`, or `data-uri` options, the consumers are enabled only if the `enable` option is specified. This is meant to allow several `enable-consumer` commands (with complex arguments and options) to be issued before committing the result. Issuing `enable-consumer` without any arguments or options commits the configuration prepared so far.

For example, this sequence of commands:

```
$ sudo -H lttn create session1
$ sudo -H lttn enable-event sched_switch -k
$ sudo -H lttn enable-consumer -C tcp://131.132.32.77:5342
$ sudo -H lttn enable-consumer -D tcp://131.132.32.77:5343
$ sudo -H lttn enable-consumer
```

is equivalent to this shorter sequence:

```
$ sudo -H lttn create session1
$ sudo -H lttn enable-event sched_switch -k
$ sudo -H lttn enable-consumer -C tcp://131.132.32.77:5342 \
  -D tcp://131.132.32.77:5343 -e
```

When the command has an argument and/or multiple options specified, the order of precedence is the `ctrl-uri` and `data-uri` options pre-empt the `set-uri` option which pre-empts the argument. For instance, these three commands are equivalent:

```
$ sudo -H lttn enable-consumer /home/<user>/traces/one \
  -U /home/<user>/traces/two -C /home/<user>/traces/three
$ sudo -H lttn enable-consumer -C /home/<user>/traces/three -e
$ sudo -H lttn enable-consumer -D /home/<user>/traces/three -e
```

Note that `enable` is unnecessary when the command has an argument, and that there can be no `data-uri` option when the `ctrl-uri` option has a `file:` argument.

Options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` command option pre-empts the `list-options` command option if it appears first on the command line.

--list-options

Shows a simple listing of the `lttng enable-consumer` options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` command option pre-empts the `help` command option if it appears first on the command line.

-s, --session <SESSION_NAME>

Applies the command to the `<SESSION_NAME>` session. Defaults to the current session. If there is no current session, an error occurs.

-k, --kernel

Applies the command to the kernel tracer. The default is to apply to both domains.

-u, --userspace

Applies the command to the user-space tracer. The default is to apply to both domains.

-e, --enable

Enables the consumer daemons. When the `set-uri`, `ctrl-uri` and/or `data-uri` options are specified while leaving the command argument-less, the consumer daemons are enabled only if the `enable` option is also specified. The consumer daemons are enabled if the command has an argument or if it has no `set-uri`, `ctrl-uri`, or `data-uri` option at all.

-U, --set-uri <URL>

Sets the `<URL>` for the `enable-consumer` destination. This will set both the data and control stream URLs for network transmission. The `ctrl-uri` and `data-uri` options pre-empt the `set-uri` option. The `set-uri` option pre-empts the command's argument. If `<URL>` is not supplied, an error occurs. **Note:** Had this command remained, the long option name would have eventually been corrected to read ‘`set-url`’ instead of ‘`set-uri`’ (see the `create` command, B.4.3).

-C, --ctrl-uri <CTRL_URL>

Sets the control stream path to <CTRL_URL>. The data-uri option can be used to specify a data stream port different from the default. The ctrl-uri option pre-empts the set-uri option and the command's argument. If <CTRL_URL> is not supplied, an error occurs. If <CTRL_URL> is a net: or a net6:, data-uri is ignored (however, if specified with a different host from <CTRL_URL>, you get an error). If <CTRL_URL> is a file: and data-uri is specified, an error occurs. **Warning:** Using tcp: or tcp6:, it is currently possible to leave the data stream port set to zero, but this will crash the consumer. **Note:** Had this command remained, the long option name would have eventually been corrected to read ‘ctrl-uri’ instead of ‘ctrl-uri’ (even though the enable-consumer --help calls it ‘ctrl-uri’; see the create command, B.4.3).

-D, --data-uri <DATA_URL>

Sets the data stream path to <DATA_URL>. The ctrl-uri option should be used to specify the control stream port as well. The data-uri option pre-empts the set-uri option and the command's argument. If <DATA_URL> is not supplied, an error occurs. Unlike the command's argument and the ctrl-uri and set-uri options, data-uri must prefix a file path with file://. <DATA_URL> cannot be a net: or net6: (an error occurs if it is). When ctrl-uri is a file:, an error occurs if data-uri is specified. When ctrl-uri is a net: or a net6:, data-uri is ignored (however, if specified with a different host from ctrl-uri, you get an error). **Warning:** Using tcp: or tcp6:, it is currently possible to set data-uri to a different host from the control stream (using the command's argument or the ctrl-uri option), but this will crash the consumer. **Note:** Had this command remained, the long option name would have eventually been corrected to read ‘data-uri’ instead of ‘data-uri’ (even though the enable-consumer --help calls it ‘data-uri’; see the create command, B.4.3).

B.4.10 enable-event

Synopsis

```
lttng [<LTTNG_OPTIONS>] enable-event
[<EVENT_NAME>[,<EVENT_NAME2>, ...] \
 (-k|-u)] [<COMMAND_OPTIONS>] [<EVENT_OPTIONS>]
```

Description

Enable and/or define tracing event(s).

Within each domain, events are distinguished by their names. The event type (tracepoint, probe, function) is an attribute of the event, not a sub-domain. Within a single domain, it is not possible to have two different-typed events bearing the same name.

A tracing event is always assigned to a channel. In the user-space domain, the same event can be assigned to multiple channels; in the kernel domain, however, an event, regardless of its type, can only be assigned to one channel. Events cannot be deleted once created (defined).

If the `channel` option is omitted, a default channel named ‘`channel0`’ is created and the event(s) is(are) added to it. However, this can happen only if the domain is ‘virgin’: if one or more channels already exist in the domain, no default channel creation occurs and you get an error instead. If the `session` option is omitted, the current session name is taken from the `.lttngrc` file if it exists (it won’t exist if no session has been created so far or if the last command was `destroy`; this will result in an error).

The `enable-event` command defines (creates) an event when the name specified does not exist yet for the domain (event names are local to each domain). See the discussion of the event list, below, to understand what constitutes an “event name”. Once an event is created, the `enable-event` command serves only to re-enable it once it has been disabled by the `disable-event` command (see Section B.4.7). If you try to redefine an event by re-issuing this command, you’ll get an “Event `<event_name>` already enabled” warning if the event was enabled; if it was disabled, however, it is *not* redefined but instead just re-enabled (with a misleading “Event `<event_name>` created” message).

The event type options (`tracepoint`, `probe`, `function`, `syscall`) should not be used to re-enable an event: they are unnecessary since the event already exists, and currently cannot be used to define a new event (same name, new type) for domain-specific reasons (the kernel does not allow event name duplication, while user-space does not support other types than `tracepoint` for now [62]).

You can freely disable and re-enable events while a trace is running (the `syscall` set of events is an exception; see the `syscall` command option, below). In earlier releases, you could not create new events once a trace had been started.

In the user-space domain, the wildcard character (‘`*`’) can be used at the end of the event name. It must be quoted (e.g. ‘`lttng enable-event -u "provider:*`’ to avoid bash expansion having unexpected effects. Using the `all` option is nearly the same as using the event name “`*`” (see the discussion of the event list, below). In the kernel domain, in previous releases you could use the wildcard by itself only (i.e. ‘`lttng enable-event -k "*"` was legal, but ‘`lttng enable-event -k "sched_*`’ would fail). With LTTng 2.3.0, the wildcard cannot be used in the kernel domain at all.

In version 2.4, the `exclude` event option is added. A command such as `enable-event -u "*" --exclude "a*,c*"` enables all events (“`*`”) except for those whose name begins with “`a`” or “`c`”.

`<EVENT_OPTIONS>` apply to all names (`<EVENT_NAME>, <EVENT_NAME2>, ...`), so unless you need exactly duplicated events, event options should be specified on a single-name basis. Multiple names make sense when enabling or disabling sets of events (typically tracepoints) together.

Event names are local to each domain, meaning the same name may be used in both domains to mean completely different events. Event names are, on the other hand, global across channels. A user-space event may be assigned to multiple user-space channels: it will create multiple record copies whenever it occurs, one in each channel (though their timestamps may differ: see *A word on timestamps* in Section 2.5). By contrast, a kernel event (of any type) cannot be assigned to more than one channel. Each kernel event will create but one record per occurrence within a session. Attempting to enable a kernel event a second time within a different channel than the first will cause an error.

Event name *shadowing* may occur in the kernel domain¹⁴: if you use a known kernel tracepoint name (such as `sched_switch`, for example) to label a kernel event of a different type (`probe` or `function`), you will not be able to enable or disable the kernel tracepoint, nor will you be able to assign it to any channel, even with the `all` option. LTTng cannot warn you against this: even if it did check your non-tracepoint event name's against the current list of kernel tracepoints, that list is susceptible to change dynamically as kernel modules are loaded and unloaded, so an event name that was not in use at the time of its creation would still shadow the tracepoint when the latter became available through module loading. The situation is even worse with the system call names: if you use *any* system call name (e.g. `sys_open`, `sys_sched_yield`, etc.), including the `compat_sys_*` names or the LTTng-reserved names `sys_unknown` and `exit_syscall`, you will no longer be able to enable system call tracing for that session, for *any* channel (you will get the “Events: Enable kernel event failed” error message) [63]. The simplest way to avoid this grief is to prefix the names you give to `probe` and `function` events with something safe like `prob_` and `func_`, for instance.

The `enable-event` command behaves somewhat differently depending on the event type. Types other than `tracepoint` (the default) are discussed under the corresponding event option, below. For tracepoints, the behaviour again varies somewhat between the kernel and user-space domains (currently, user-space events can only be of the `tracepoint` type). In the kernel domain, the `<EVENT_NAME>` arguments are immediately validated against the kernel’s list of tracepoint events (as obtained by the `list -k` command).

In the user-space domain, however, event names are initially *unbound*: the session daemon treats them as logical entities with attributes such as enabled/disabled, logging level, and filter. When a new instrumented user-space process registers itself (and its events) with the session daemon, the daemon binds the event declarations *using only the names*. While a session is running, expect the bindings of user-space event names to change dynamically as processes register and unregister themselves with the session daemon. At any given moment, a user-space event name could be bound to none, one, or several processes. Further, it is perfectly fine for two different processes to use completely different payload definitions for the same event name: this will not confuse `babeltrace` (although it may confuse the analyst). Using dynamically loaded tracepoint providers, a single process could generate events using one payload definition for a while, then disconnect itself from the session daemon and reconnect using another payload definition for the same event name.

¹⁴ Shadowing does not occur in the user-space domain because there is just one type of event.

One problem with the forthcoming support for user-space probe-like events is that non-instrumented processes do not register themselves with the session daemon, and having the daemon watch new process launches in order to insert probes previously defined would incur considerable overhead. How this desirable feature is eventually implemented will be interesting.

The user-space event list

In user-space, the session daemon stores each enable-event command in a sort of list. Kernel event management is simpler because it is immediate, on the one hand, and does not admit wildcards, on the other. The user-space event list has a distinct entry for each <EVENT_NAME> and each entry holds an enabled/disabled Boolean flag. All the entries applicable to an event (all those for which the <EVENT_NAME> matches, considering any wildcards) are ORed together to decide whether or not to log a particular event record. Another way to look at the list is to treat it as a set of production rules: as long as at least one enabled rule applies, the event gets logged.

Entries in the list are further distinguished by their filtering options (loglevel, loglevel-only, filter and the forthcoming exclude). To illustrate this point, consider the following sequence of commands (edited for legibility):

```
$ lttnng enable-event -u -a
$ lttnng disable-event -u -a
$ lttnng enable-event -u -a --loglevel warning
$ lttnng list <session>
Events:
* (type: tracepoint) [disabled]
* (loglevel: TRACE_WARNING (4)) (type: tracepoint) [enabled]
```

Two same-name entries have different filters if the filter expressions are different. Filter expressions are compared once compiled to pseudo-code; the comparison is thus not sensitive to extra white space nor extra parentheses: "value > 0", "(value > 0)" and "value > 0" are the same, but "value > 0" and "0 < value" are different.

The "*" entry is special: if it does not already exist, it is created by the enable-event --all command. It is otherwise treated *as a separate entry*, however. For instance, if you have entries like this:

```
$ lttnng list <session_name>
Events:
* (type: tracepoint) [enabled]
abcd (type: tracepoint) [enabled]
```

Then the disable-event "*" command would disable just the "*" entry, while the disable-event --all command would truly disable all entries.

The disable-event command does not recognise any event option (loglevel, loglevel-only, filter, tracepoint, probe, function, syscall): it disables all entries in the list with matching <EVENT_NAME>s, *ignoring* any wildcards (that is to say, it treats them as literal characters). It thus acts as a kind of master switch for the set of entries bearing that <EVENT_NAME>. Within that set, toggling some of them off and others on will usually require a disable-event command followed by one or more enable-event commands. Here are examples:

```
$ lttng enable-event -u abcd
UST event abcd is enabled in channel ch0
$ lttng enable-event -u abcd --loglevel warning
UST event abcd is enabled in channel ch0 for loglevel TRACE_WARNING (4)
$ lttng enable-event -u abcd --filter "value > 100"
UST event abcd is enabled in channel ch0 with filter 'value > 100'
$ lttng list session
Events:
    abcd (type: tracepoint) [enabled]
    abcd (loglevel: TRACE_WARNING (4)) (type: tracepoint) [enabled]
    abcd (type: tracepoint) [enabled] [with filter]
$ lttng disable-event -u abcd
$ lttng list session
Events:
    abcd (type: tracepoint) [disabled]
    abcd (loglevel: TRACE_WARNING (4)) (type: tracepoint) [disabled]
    abcd (type: tracepoint) [disabled] [with filter]
$ lttng enable-event -u abcd --loglevel warning
$ lttng list session
Events:
    abcd (type: tracepoint) [disabled]
    abcd (loglevel: TRACE_WARNING (4)) (type: tracepoint) [enabled]
    abcd (type: tracepoint) [disabled] [with filter]
$ lttng enable-event -u abcd --filter "value > 200"
UST event abcd is enabled in channel ch0 with filter 'value > 200'
$ lttng list session
Events:
    abcd (type: tracepoint) [disabled]
    abcd (loglevel: TRACE_WARNING (4)) (type: tracepoint) [enabled]
    abcd (type: tracepoint) [disabled] [with filter]
    abcd (type: tracepoint) [enabled] [with filter]
```

Currently, the list command does not detail the filter attached to each event [64]. The last two lines in the example above refer respectively to the 'value > 100' filter (the disabled event entry) and the 'value > 200' filter (the enabled event entry).

The existence of this event list sometimes leads to odd effects such as rejection messages being duplicated. For instance, if one enables "*" and "sample_component:message" (using either a single enable-event command or two separate commands) and a trace provider tries to register such an event with an unsupported payload (i.e. a long double field, see Section 4.1.2.3), the session daemon will reject the registration twice.

Options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` command option pre-empts the `list-options` command option if it appears first on the command line.

--list-options

Shows a simple listing of the `lttng enable-event` options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` command option pre-empts the `help` command option if it appears first on the command line.

-s, --session <SESSION_NAME>

Applies the command to session `<SESSION_NAME>`. Defaults to the current session. If there is no current session, an error occurs.

-c, --channel <CHANNEL_NAME>

Applies the command to channel `<CHANNEL_NAME>`. This option can be omitted only while the session has a single, default channel, “`channel0`”. If “`channel0`” was explicitly enabled, it is *not* considered a default channel. Once you have created a channel explicitly within a domain, the `<CHANNEL_NAME>` must be specified or else you get a “`Non-default channel exists within session: channel name needs to be specified`” error.

-a, --all

Enables all events. `<EVENT_NAME>` is ignored and may be omitted. The event type may be specified by the appropriate command option. The command will fail if the event type is unsupported by the domain.

In the kernel domain, all the unassigned tracepoints are assigned to the specified channel (or the default channel if the domain is virgin). The syscalls are *also* assigned to the channel if so far unassigned. Specifying the `tracepoint` or `syscall` option alongside the `all` option assigns just the tracepoints or syscalls, respectively, to the channel. Specifying another event type (`probe` or `function`) fails, even though they are supported: it’s just that “`all`” is meaningless for those types. Note that if you enable all events in one channel and then enable all events in another, the other channel is created but will hold no events *and you get no warning to this effect*.

In the user-space domain, enables the “`*`” tracepoints of the specified channel —note that this is *completely different* from the `disable-event` command’s `all` option. Specifying any event type other than `tracepoint` fails, because the user-space domain supports just that type for now.

-k, --kernel

Applies the command to the kernel tracer (there is no default). A domain must be specified. The `kernel` option pre-empts the `userspace` option.

-u, --userspace

Applies the command to the user-space tracer (there is no default). A domain must be specified. The `kernel` option pre-empts the `userspace` option.

Event options

Do not specify multiple event types (`tracepoint`, `probe`, `function`, `syscall`); if you do, all but the last type option will be ignored.

--tracepoint

Tracepoint event (default). The tracepoint enabled is specified by the `<EVENT_NAME>`. The user-space tracer supports wildcards at the end of the `<EVENT_NAME>` string (e.g. `"sample_component:/*"`); the kernel tracer used to accept unqualified wildcards (i.e. `"*"`) and explicit event names (e.g. `"sched_switch"`), but now accepts explicit event names only. This type of event is currently the only one supported for both the kernel and user-space (and the only one provided by user-space).

Kernel tracepoints are all at loglevel `TRACE_EMERG` (see Section 4.1.2.4).

--probe (<addr> | <symbol> | <symbol>+<offset>)

Dynamic probe (`kprobe`). The probed address must be specified numerically (`addr`) or symbolically (`symbol`) or you'll get an ‘Undefined command’ error. The `addr` and `offset` can be octal (`0NNN...`), decimal (`NNN...`) or hexadecimal (`0xNNN...`). Offsets beyond $2^{64}-1$ (`0xFF FF FF FF FF FF FF FF` hexadecimal, 4 294 967 295 decimal) are reduced to that value (no warning is issued). This type of event is currently supported by the kernel only (work is on-going to bring this to user-space as well).

Example

```
# lttng enable-event -k kernel_sys_open --probe sys_open+0x0
```

The `kprobe` mechanism has the limitation that any `symbol` used can only be applied to a code section: you cannot use a data `symbol`. LTTng returns an “Enable kernel event failed” error if the `symbol` is unrecognised or the address disallowed. In addition to data `symbols`, the kernel won’t allow you to install a `kprobe` (or `kretprobe`) in the code that implements `kprobes` (e.g. symbols such as `do_page_fault` and `notifier_call_chain`), because that would trigger an infinite loop.

You can probe the same address using multiple events, in the same channel or in different ones, the only restriction being that each named kernel event cannot be assigned to more than one channel (nor redefined). The list of eligible kernel code symbols can be obtained by the following command: ‘cat /proc/kallsyms | grep “ [Tt] ”’. It typically contains around 40,000 entries.

The event records that this generates contain a single payload field, `ip`, the instruction pointer (IP) register value (i.e. the probe’s address).

--function (<addr> | <symbol> | <symbol>+<offset>)

Dynamic function entry/return probe (`kretprobe`). This command actually creates (enables) two events, `<EVENT_NAME>_entry` and `<EVENT_NAME>_return`, occurring upon function entry and return, respectively. The `addr`, `symbol` and `offset` obey the same rules as with the `probe` event option. This type of event is currently supported by the kernel only. This option is also handled by the `kprobe` mechanism, so the `symbol` limitations mentioned for the `probe` event option also apply. You’ll get the same error message if you specify an unrecognised or disallowed probe address.

The event records that this generates contain two payload fields, `ip` and `parent_ip`, the address of the probe and an undocumented instruction pointer value.

--syscall

System call event. By definition, this is supported by the kernel only—but see Annex J for an equivalent user-space facility. Currently, LTTng does not allow per-system call selection, so this option *must* be used with the `all` option. If you previously enabled all kernel tracepoints, enabling `syscalls` causes the unnecessary warning “Kernel events already enabled” [65] (this is fixed in version 2.4). Also, you will *not* be able to disable them with `disable-event` (issuing ‘`lttng disable-event -k syscalls`’ will fail with “Event not found”). To disable system call events, your only recourse is to disable the channel you’ve assigned these events to (see the `disable-channel` command, Section B.4.5). System call events can only be assigned to one channel. Another side effect of enabling system call events is that you will no longer be able to use any system call name (e.g. `sys_open`, `compat_sys_open`, `sys_sched_yield`, etc.), including the LTTng-reserved names `sys_unknown` and `exit_syscall`, for `probe` or `function` events, in any channel [63].

The trace will register each system call entry event under the system call’s name, along with its arguments. For example, the `sys_recvmsg` system call has this prototype (declared in `include/linux/syscalls.h`):

```
long sys_recvmsg(int fd, struct msghdr __user *msg, unsigned flags)
```

Trace event records named `sys_recvmsg` will thus have the fields `fd`, `msg`, and `flags`. Unrecognised system calls are reported as `sys_unknown` events with two fields: `id`, the system call number, and `args[6]`, an array of unsigned long integer parameters [66] —system calls have at most six parameters. This may happen upon upgrading the kernel if the new kernel adds one or more new system calls. LTTng's system call headers need to be updated and LTTng recompiled; the procedure is described in `lttng-modules/instrumentation/syscalls/README`.

System call exit events all bear the `exit_syscall` name (this avoids a name collision with the extant system call `sys_exit`), and have a single field, `ret`, the return value. If no events are lost, every other record will be an `exit_syscall`.

--loglevel <NAME>

Sets a tracepoint loglevel upper limit: among the events, only those of levels 0 through `<NAME>` will be logged. Currently, loglevels are supported for the user-space domain only. By default, no loglevel filtering occurs. The loglevels are listed in Section 4.1.2.4. The "TRACE_" or "TRACE_DEBUG_" prefix can be omitted (e.g. "SYSTEM" for "TRACE_DEBUG_SYSTEM", "DEBUG" for "TRACE_DEBUG"; no other abbreviations are recognised) and the `<NAME>` is not case-sensitive (e.g. "SYSTEM", "system" or "SyStEm" will do).

In the kernel domain, the `loglevel` and `loglevel-only` options are ignored if the event type is `syscall`, but an error occurs if it is `probe` or `function`. If the `all` option is used, the interface gives the impression it accepted loglevel filtering, but it is actually ignored [67]. This is fixed in version 2.4.

--loglevel-only <NAME>

Sets a tracepoint loglevel match requirement: among the events, only those of level `<NAME>` will be logged. Currently, loglevels are supported for the user-space domain only. By default, no loglevel filtering occurs.

Only the last `loglevel` or `loglevel-only` option specified is used (`loglevel` and `loglevel-only` thus pre-empt each other). In the kernel domain, `loglevel` and `loglevel-only` are ignored if the `all` option was set; otherwise an error occurs.

Unfortunately, currently both the `loglevel` and `loglevel-only` options yield the same confirmation message and are displayed the same way by the `list <session>` command, so you can't readily tell which loglevel filtering setting is in use [68] (this is fixed in version 2.4).

-f, --filter <expression>

Sets a filter on a newly enabled event. The `all` command option pre-empts the `filter` event option. If `<expression>` contains spaces, it must be quoted using single ('`<expression>`') or double quotes ("`<expression>`"), or each space must be escaped ('\ '). Single quotes are easiest, since they wholly prevent command-line shell expansion. Malformed `<expression>`s are rejected with an “Invalid filter bytecode” error.

The Boolean `<expression>` is evaluated for each event against the event’s field names (see the `lttng list` command, B.4.11, and the `TRACEPOINT_EVENT` macro, 4.1.2.3; the field names are the `_items` of the `ctf_*` calls of the `TP_FIELDS` macro) using a small sub-set of standard C syntax that includes the Boolean operators (`!`, `&&`, `||`), comparison operators (`==`, `!=`, `>`, `>=`, `<`, `<=`), unary arithmetic operators (`+`, `-`) and parentheses (nesting is allowed only for the logical operators but is buggy [69]; this is fixed in version 2.4). It controls whether or not recording occurs. As usual with C, anything that evaluates to non-zero is true. If the filter fails to link with the event within the traced domain, the event is also discarded. A filter links successfully with an event description only if all of the field names appearing in the `<expression>` are present in the event’s payload —the remaining fields of the event (e.g. `timestamp`, `trace:hostname`, `cpu_id`, etc.) are not eligible for filtering, but context fields are (see below).

Unfortunately, currently the `list <session>` command simply states that the event is “[with filter]”, so you can’t readily tell which filter expression is in use [64]. This is particularly problematic when the `<EVENT_NAME>`s are the same.

LTTng compiles the filters into byte code which is stored by the session daemon. When an instrumented application connects to the session daemon, its tracepoint provider receives the byte code as part of the enabling of its events; that byte code is executed by the tracepoint provider when the event is triggered, using a virtual machine supplied by the `liblttng-ust` dynamic library. However, byte code being inherently slower, filters should be used sparingly.

In particular, a tracepoint’s arguments will be evaluated before the filter expression itself (this is necessary since the filter expression may depend on the argument values), whether the event passes the filter test or not. This means computing-intensive argument expressions should be avoided as much as possible. (LTTng version 2.7 intends to introduce some filtering options that should alleviate this problem, by allowing user-space channels to be restricted to certain user or process ID values [70])

Expression examples

```
intfield > 500 && intfield < 503  
(stringfield == "test" || intfield != 10) && intfield > 33  
doublefield > 1.1 && intfield < 5.3
```

The context, which the `add-context` command (Section B.4.1) can add to the event records of any or all channels, is accessible to the filters —regardless of whether or not it has been added to the event’s channel. In other words, you need not `add-context` to a channel in order to use the context in a filter. Context is accessed using the `$ctx` symbol:

```
$ctx.vpid >= 4433 && $ctx.vpid < 4455
```

Wildcards are allowed at the end of comparison strings (they match any sequence of characters, including the empty string):

```
seqfield1 == "te*"
```

In string literals, the escape character is a backslash ('\'). If you need to compare to a backslash or a literal asterisk ('*'), you will need to escape them ('*' for '*', '\\\' for '\'). You may also need to guard against the command-line shell expanding the <expression>. For example, this <expression>:

```
$ctx.procname == "sample_*"
```

Can be written in an `lttng enable-event` command line in several ways:

```
--filter '$ctx.procname == "sample_*"'
--filter "\$ctx.procname == \"sample_*\""
--filter \$ctx.procname\ ==\ \"sample_*\"
--filter \$ctx.procname==\"sample_*\""
```

Be careful with field names: a misspelled field name in the <expression> won't cause an error, but the events you expect will be discarded because of the mismatch. As an aside, note that testing the `$ctx.procname` is not very reliable: it will be "fooled" by hard or symbolic links.

Currently, the `filter` option is only implemented for the user-space tracer. To find out what the available fields are for a user-space event, use the `list` command's `fields` and `userspace` options (see Section B.4.11) while a provider for that event is active.

Kernel event filters

In the near future filters will become available for the kernel domain, and one will be able to use the `list --fields -k` command to find out what the available fields are for each kernel event. As explained in the `add-context` command, Section B.4.1, the names of the `perf` fields are already transformed to make them compatible with use in an <expression>.

Without a `list --fields -k` command, finding out what the available fields are for a kernel event must be done through manual research. You can find out after the fact by using `babeltrace` in order to list the trace records' field names (the `event.fields`). Or you can look at the system's header file (<system>.h) in the `lttng-modules` source code, found in the `/usr/src/lttng-modules/instrumentation/events/lttng-module` folder (the `TP_STRUCT__entry` macro of the `TRACE_EVENT`; see Section 4.6.1.3.3).

The debugfs tracing service (see Topical Note A.6) usually also defines the same tracepoints, but this is less reliable. The debugfs tracepoint field descriptions are in the `print fmt` part of the `format` file, found in the `/sys/kernel/debug/tracing/events/<system>/<event_name>` folder. Some LTTng events are omitted from this set of files and there are some occasional minor differences. Finally, you can look at the tracepoint system's header file in the kernel headers or source code (`linux-headers` and `linux-source` packages, respectively). The `<system>.h` file is found in the `include/trace/events` folder. Although the `<system>.h` file of `lttng-modules` is derived from the corresponding Linux `<system>.h` file, there may again be slight differences (see Section 4.6.1.1).

For instance, these are the `sched_switch` event field names given by the various sources:

```
babeltrace -f all -n all and  
lttng-modules/instrumentation/events/lttng-module/sched.h:  
  
prev_comm, prev_tid, prev_prio, prev_state, next_comm, next_tid, next_prio  
  
/sys/kernel/debug/tracing/events/sched/sched_switch/format,  
linux-headers/include/trace/events/sched.h and  
linux-source/include/trace/events/sched.h:  
  
prev_comm, prev_pid, prev_prio, prev_state, next_comm, next_pid, next_prio
```

Note how `prev_pid` and `next_pid` are listed as `prev_tid` and `next_tid` by LTTng.

B.4.11 list

Synopsis

```
lttng [<LTTNG_OPTIONS>] list [<SESSION> [<SESSION_OPTIONS>]] [-  
k|-u] \  
[<COMMAND_OPTIONS>]
```

Description

List tracing session information. With no arguments, it just lists the user's available tracing sessions. Note that (see Section 2.2.1) for a user belonging to the `tracing` group, this option will list only his own sessions, whereas for the root user (`sudo lttng list`) this option will list all sessions managed by the root session daemon, regardless of which user created them.

Without a <SESSION>, the kernel option will list all available kernel events (except the syscall events) and the userspace option will list all available user-space events from currently registered applications. Here is an example of 'lttng list -u':

```
PID: 3981 - Name: /tmp/lttng-ust/tests/hello/.libs/lt-hello
    ust_tests_hello:tptest_sighandler (type: tracepoint)
    ust_tests_hello:tptest (type: tracepoint)
PID: 7448 - Name: ./sample
    sample:message (loglevel: TRACE_WARNING (4)) (type: tracepoint)
```

The name reported is actually the command line used to launch the process. The first part of each event name is the trace provider's name (in this case `ust_tests_hello`; see `TRACEPOINT_PROVIDER` in 4.1.2.2); the second part is the actual event name (here `tptest_sighandler` and `tptest`). The `loglevel` is reported when defined. You can enable or disable any event listed by using the name thus obtained (e.g. `lttng enable-event ust_tests_hello:tptest -u --channel <channel_name>`).

User-space events exist (will be reported by `lttng list -u`) only as long as the application that emits them is running (more accurately, as long as it is registered with the session manager). You can, of course, define events for a tracing session (using `lttng enable-event`) well in advance of the event registering with the session manager. The session manager simply matches the names of any newly declared events (declarations issued by the trace provider as the process starts up) with the ‘shopping list’ of each currently running session.

The kernel event list is quite extensive and varies with the specific modules included in your installation. Here are a few samples from that list (a full list would run in the vicinity of 260 entries):

balance_dirty_pages	net_dev_xmit
bdi_dirty_ratelimit	netif_rx
block_sleeprq	power_domain_target
clock_set_rate	put_swap_token
consume_skb	rcu_utilization
cpu_frequency	regcache_sync
cpu_idle	regmap_hw_read_done
disable_swap_token	regulator_enable
ext3_free_inode	regulator_set_voltage
global_dirty_state	replace_swap_token
gpio_value	rpm_suspend
hrtimer_start	sched_switch
irq_handler_entry	scsi_dispatch_cmd_start
itimer_expire	scsi_eh_wakeup
jbd_start_commit	signal_generate
jbd2_start_commit	skb_copy_datagram_iovec
kfree	snd_soc_bias_level_start
kmalloc	sock_exceed_buf_limit
kmem_cache_free	softirq_raise
kvm_ioapic_set_irq	timer_init
lttng_statedump_start	udp_fail_queue_rcv_skb
machine_suspend	update_swap_token_priority
mm_page_alloc	wbc_writepage

module_load	workqueue_queue_work
napi_poll	writeback_exec

With a <SESSION>, the `list` command displays the details of the session including its state (active or inactive), the trace file path, the channels of each domain and their state (enabled or disabled), the events of each channel and more. You can pare the list of channels down to a specific domain's channels by adding the <domain> option (e.g. `list <SESSION> --kernel` lists only the kernel channels and their details). Specifying the `channel <CHANNEL>` option lists just that channel and its details (in combination with the <domain> option or not).

Options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` command option pre-empts the `list-options` command option if it appears first on the command line.

--list-options

Shows a simple listing of the `lttng list` options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` command option pre-empts the `help` command option if it appears first on the command line.

-k, --kernel

Selects the kernel domain. Without a <SESSION>, lists all available kernel events (except the systemcall events). With a <SESSION>, restricts the list of channels to the kernel channels.

-u, --userspace

Selects the user-space domain. Without a <SESSION>, lists all available user-space events from registered applications. With a <SESSION>, should restrict the list of channels to the user-space channels but this is currently ignored [71] (this is fixed in version 2.4).

-f, --fields

Lists the fields of currently registered events. This is ignored if <SESSION> is specified. This option is currently for user-space events only: it is ignored if the `userspace` option is not set or if the `kernel` option is set (so you cannot use it to obtain the kernel tracepoint event fields). The latter is not really a problem since the `enable-event` command's `filter` option is also currently restricted to user-space. It is expected that the user-space-only restriction will be lifted for this command option shortly before (or at the same time as) it is also lifted for the `enable-event` command's `filter` option.

The display identifies the process which registered the user-space event(s), the events themselves, and finally their fields, as shown in the example below. Each process is identified by both its ID and its “name” (that is to say, the command-line used to launch it). The events are identified by their full names (provider name and event name) and specify their current logging level and type. Currently, the event type can only be `tracepoint`. Each field is named and its data type specified. An event can have no fields, in which case no ‘`field:`’ line will appear. (Note that the fields are in reverse order: the field defined first by the `TP_ARGS` macro appears last in the list `-f` listing)

```
UST events:
```

```
-----
```

```
PID: 6221 - Name: ./application
    provider:event (loglevel: TRACE_WARNING (4)) (type: tracepoint)
        field: floatfield (float)
        field: otherfloatfield (float)
```

Session options

-c, --channel <NAME>

Lists only the details of the channel `<NAME>` of the `<SESSION>`. Each channel `<NAME>` of each domain is listed unless a domain option (`kernel` or `userspace`, but see [71]) is also specified. This is ignored if `<SESSION>` is not specified. By default, `list <SESSION>` lists all of the session’s channels.

-d, --domain

Lists the domains available within the `<SESSION>`. If `<SESSION>` is not specified, this is ignored. Specifying `domain` prevents channels from being listed (the `channel`, `fields`, `kernel`, and `userspace` options are ignored).

B.4.12 set-session

Synopsis

```
lttng [<LTTNG_OPTIONS>] set-session <SESSION_NAME>
[<COMMAND_OPTIONS>]
```

Description

Set the current session name to `<SESSION_NAME>`.

This changes the session name in the `.lttngrc` file. No validation is done: you get no error or warning if you set your current session name to one which is not legal, which does not exist or which is not accessible to you (another user's root-level session, for instance). The only error possible is if you omit to specify a `<SESSION_NAME>` at all. Validation will occur when you try to actually do something to or with the session.

Options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` command option pre-empts the `list-options` command option if it appears first on the command line.

--list-options

Shows a simple listing of the `lttng set-session` options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` command option pre-empts the `help` command option if it appears first on the command line.

B.4.13 `snapshot`

Synopsis

```
lttng [<LTTNG_OPTIONS>] snapshot [<OPTIONS>] <ACTION> [<OPTIONS>]
```

Description

This command was introduced by LTTng 2.3.0-rc1 (July 17, 2013). This command manages sessions running in snapshot mode (a.k.a. flight recorder mode). It serves to trigger snapshot output and to configure said output.

The `lttng create --snapshot` command automatically sets up the buffers in `overwrite` mode for “flight recording”: the intent is that the trace will be used to generate snapshots exclusively. Snapshot-mode tracing sessions and “normal” tracing sessions are mutually exclusive, whereas normal tracing sessions can mix and match channels in `discard` or `overwrite` mode. The session *must* have been created with the `no-output` or `snapshot` options (see Section B.4.3), otherwise none of the `<ACTION>`s will succeed. If you merely specify `no-output` but leave the session’s channels in `discard` mode, the buffers will fill up and the trace will then stop. The `snapshot` command can later be used to save their contents to persistent storage, but there is currently no way to flush the buffers in order to proceed further [72]. An action to that effect could be added to the `snapshot` command in the future.

An <ACTION> (among the four available actions detailed below) must be specified. Only the first <ACTION> is handled: any remaining ones on the command line are ignored. The most important <ACTION> is `record`, which actually copies the buffers to a local or remote trace snapshot. The remaining <ACTION>s simply manage the `record`'s output parameters.

The command options (<OPTIONS> in the synopsis line) can appear before or after the <ACTION>. The `help`, `list-commands` and `list-options` options pre-empt the <ACTION>. Which of the remaining command options have any effect depends on the <ACTION> as detailed below. If the <ACTION> accepts or expects an argument, the argument cannot precede the <ACTION>.

Here is a simple example of a snapshot-mode tracing session:

```
$ lttnng create --snapshot
    [ enable some events, configure the trace ]
$ lttnng start
    [ do something, generate activity on the system ]
$ lttnng snapshot record
    [ do more stuff... ]
$ lttnng stop
$ lttnng destroy
```

Each `lttnng snapshot record` command records a snapshot of the current buffer state (the set of trace events in the current buffers). The `lttnng snapshot record` command can be invoked while tracing is started or paused. You will get an error if you try to take a snapshot before the trace is even started.

The buffers are saved as a complete trace, as a subfolder of the session folder. The snapshot subfolder is named <output>-<timestamp>-<number>, where <output> is the snapshot output parameter set's name (initial default `snapshot-1`), the <timestamp> is the usual date and time sequence (when the `lttnng snapshot record` command was issued), and <number> is a snapshot sequence number that starts at zero. For instance, two successive `lttnng snapshot record` commands would create the subfolders `snapshot-1-20130721-141838-0` and `snapshot-1-20130721-141840-0` (the trailing sequence number is meant to distinguish between snapshots taken within the same one-second time window, but this currently does not work [73]). The `name` option can be used to override the snapshot output name (e.g. '`lttnng snapshot record -n <somename>`' would name the folder <somename>-<timestamp>-<number>). Babeltrace should be given either the session folder or an individual snapshot folder as input, depending on whether you want the snapshots merged or not.

The `record` action notes the readable extent of the trace buffers at its outset, and tries to commit that sequence of records to storage. There are `num-subbuf` sub-buffers in a channel’s ring buffer, and each sub-buffer is tagged with a 64-bit wide sequential number (an absolute index of sorts). At any given time, each sub-buffer can only be in one of to access modes: either “read” (a consumer/reader is copying its contents to storage) or “write” (one or more tracers/writers are writing one or more event records into it). The consumer skips sub-buffers that are busy being overwritten, and the tracers skip sub-buffers that are busy being read [74]. The `record` action sweeps the sub-buffers running between the starting and ending indices determined at the outset, skipping any busy sub-buffers and any “missing” sub-buffers. A sub-buffer in the span can go missing if the writers overtake the consumer and overwrite some sub-buffers ahead of it (overwritten sub-buffers will have the wrong absolute indices). The snapshot trace may thus have some gaps in its timestamps. Note in particular that the consumer will not keep going past its ending index, even if tracers have filled them with the correct “next” sub-buffers while it was going around the ring, storing the snapshot. The action’s `max-size` option and the channels’ `tracefile-size` and `tracefile-count` options may also limit the size of the generated snapshot.

Core dump handler

Since version 2.3.0, `lttng-tools/extras/core-handler` contains a demonstration of an LTTng-aware core dump handler script. The Linux kernel uses the `/proc/sys/kernel/core_pattern` file as a target whenever a core dump occurs. As detailed in the `man core` pages, the `core_pattern` can be resolved to a storage path or to a pipe; in the latter case, the core dump data are piped to the indicated executable. When installed, the LTTng core dump handler script backs up the original `core_pattern` and substitutes itself as the target. When a core dump occurs, the script checks if the LTTng root session daemon is running and, if that is indeed the case, issues an `lttng snapshot record -s coredump-handler` command before directing the core dump data to a predetermined storage location. This last step could fairly easily be replaced with an invocation of the previous `/proc/sys/kernel/core_pattern` contents, thus setting up a core dump tool chain. The `coredump-handler` session (another session name could easily be used) is expected to have been created in `snapshot` mode by the root session daemon, but can otherwise be configured arbitrarily —for instance, it need not be tracing kernel events at all, but could be tracing user-space events instead (it could also be tracing both domains, of course).

Options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the <ACTION> and any remaining options are ignored. The `help` command option pre-empts the `list-*` command options if it appears first on the command line.

--list-commands

Shows a simple listing of the `lttng` snapshot actions and then quits. When this option is specified, the `<ACTION>` and any remaining options are ignored. The `list-commands` command option pre-empts the `help` and `list-options` command options if it appears first on the command line.

--list-options

Shows a simple listing of the `lttng` snapshot options and then quits. When this option is specified, the `<ACTION>` and any remaining options are ignored. The `list-options` command option pre-empts the `help` and `list-commands` command options if it appears first on the command line.

-s, --session <SESSION_NAME>

Applies to all actions. Applies the `<ACTION>` to session `<SESSION_NAME>`. Defaults to the current session. If there is no current session, an error occurs.

-n, --name <NAME>

For the `record` action, name the snapshot `<NAME>` (defaults to the current output set name); for the `add-output` action, name the output set `<NAME>` (defaults to `snapshot-<ID>`). Ignored by the remaining actions.

-m, --max-size <SIZE>

Applies to the `add-output` and `record` actions, ignored by the others. Limits the snapshot to no more than `<SIZE>` bytes (excluding the metadata file). Only the first `<SIZE>` bytes of the buffer will be saved to the snapshot. `<SIZE>` can be up to $2^{64}-1$ (9,223,372,036,854,775,807, hexadecimal 7FFF FFFF FFFF FFFF, 16 exbibytes minus one byte) inclusive. The suffixes K, M, or G may be used to specify respectively kibibytes, mebibytes, or gibibytes (the suffix k is also taken to mean kibi). A `<SIZE>` prefixed with `0x` is taken to be in hexadecimal, while a `<SIZE>` prefixed with a leading zero is taken to be in octal.

Currently, `lttng` rejects hexadecimal `<SIZE>` values that contain the hexadecimal digits A through F, and does not recognise the `0x` hexadecimal prefix (which is standard C) [57]. There is also a rounding problem [58]: if you specify a $2^{64} > <SIZE> > 2^{63}$, `lttng` will “round it up” to 1 instead of rejecting it. The latter bug is fixed in version 2.4.

-C, --ctrl-url <CTRL_URL>

Applies to the `add-output` and `record` actions, ignored by the others. Sets the control stream path to `<CTRL_URL>`. This option and its complement the `data-url` option function precisely as with the `create` command (Section B.4.3, in particular the *URL format* subsection). If `<CTRL_URL>` is not supplied or is a `file:` or `ctrl-url` is not accompanied by `data-url`, an error occurs —even when the `<ACTION>` actually ignores this option. If `<CTRL_URL>` is a `net:` or a `net6:`, `data-url` must be identical.

```
$ lttnng snapshot add-output -C net://131.132.32.77:5342:5343 \
  -D net://131.132.32.77:5342:5343
```

-D, --data-url <DATA_URL>

Applies to the `add-output` and `record` actions, ignored by the others. Sets the data stream path to `<DATA_URL>`. The `ctrl-url` option must be used to specify the control stream port as well. If `<DATA_URL>` is not supplied or is a `file:` or `data-url` is not accompanied by `ctrl-url`, an error occurs —even when the `<ACTION>` actually ignores this option. Unlike the `add-output` or `record` action's argument and the `ctrl-url` option, `data-url` must prefix a file path with `file://`. `<DATA_URL>` cannot be a `net:` or `net6:` (an error occurs if it is). When `ctrl-url` is a `net:` or a `net6:`, `data-url` is ignored (however, if specified with a different host from `ctrl-url`, you get an error).

Actions

```
add-output [-m <SIZE>] [-s <SESSION_NAME>] [-n <OUTPUT_NAME>]
(<URL> | -C <URL> -D <URL>)
```

Sets up snapshot output parameters for a session. All options are applicable; `<URL>` pre-empts `ctrl-url` and `data-url`, but either `<URL>` or the `ctrl-url`, `data-url` pair must be specified. The `<OUTPUT_NAME>` is for future reference by the `del-output` and `record` actions. If `max-size` is not specified, `<SIZE>` is set to `-1`, which means the snapshot is unlimited in size.

Currently, only one output object may be created per session; eventually several output objects will be allowed for each session, each output receiving a copy of the snapshot data. This is the reason why each snapshot output parameter set may be named and also receives a numeric identifier. Until this time, you must `del-output` before you can `add-output` again. Note that the `lttnng create --snapshot` command creates the snapshot session with its output already configured by default: you must `del-output` before adding a custom output. In contrast, a session created using merely the `no-output` option has *no* output set by default: you will have to add one before you can use the `record` action successfully.

Currently, the output set IDs start at 1 and increase by one every time a new output set is specified. When the `<OUTPUT_NAME>` is not specified, the output set's name is `snapshot-<ID>`. Example:

```

$ lttng create --snapshot snappy
Session snappy created.
Default snapshot output set to:
  /home/<username>/lttng-traces/snappy-20130917-163254
Snapshot mode set. Every channel enabled for that session will be
  set in overwrite mode and mmap output.
$ lttng snapshot list-output
Snapshot output list for session snappy
[1] snapshot-1:
  /home/<username>/lttng-traces/snappy-20130917-163254
$ lttng snapshot del-output 1
Snapshot output id 1 successfully deleted for session snappy
$ lttng snapshot list-output
Snapshot output list for session snappy
  None
$ lttng snapshot add-output ~/lttng-traces/snappy
Snapshot output successfully added for session snappy
[2] : /home/<username>/lttng-traces/snappy (max-size: -1)
$ lttng snapshot list-output
Snapshot output list for session snappy
[2] snapshot-2: /home/<username>/lttng-traces/snappy

```

Note the highlighted line. This is a bug, and it is fixed in version 2.4 of LTTng [75]. The `add-output` command failed to report the output's default name, like this:

```

$ lttng snapshot add-output ~/lttng-traces/snappy
Snapshot output successfully added for session snappy
[2] snapshot-2:
  /home/<username>/lttng-traces/snappy (max-size: -1)

```

del-output <OUTPUT_ID> | <OUTPUT_NAME> [-s <SESSION_NAME>]

Deletes a snapshot output parameter set, identified by its `<OUTPUT_ID>` or `<OUTPUT_NAME>`. The `session` option is applicable; the others are ignored —even the `name` option.

list-output [-s <SESSION_NAME>]

Lists the snapshot output parameter sets of a session. The `session` option is applicable; the others are ignored. Here is an example output:

```

$ lttng snapshot list-output
Snapshot output list for session <session>
[1] snapshot-1:
/home/<user>/lttng-traces/<session>-<date>-<time>
$ lttng snapshot del-output 1
$ lttng snapshot list-output
Snapshot output list for session <session>
None

```

**record [-m <SIZE>] [-s <SESSION_NAME>] [-n <OUTPUT_NAME>]
[<URL> | -C <URL> -D <URL>]**

Captures a session’s current buffers (for all domains), using the current snapshot output parameter set. All options are applicable, each overriding the corresponding attribute of the current snapshot output parameter set. Thus you can use <URL> or the `ctrl-url` and `data-url` options (<URL> pre-empts `ctrl-url` and `data-url`) to send the snapshot elsewhere, you can use `name` to specify a different snapshot name, and you can use `max-size` to set the snapshot’s maximum size. If `max-size` is not specified, <SIZE> is set to `-1`, which means the snapshot is unlimited in size.

Currently, several bugs manifest themselves as `lttng record` claiming a successful snapshot while not committing anything to persistent storage. In one case, this happens if the session was created using merely the `no-output` option and neither <URL> nor the `ctrl-url` and `data-url` options are specified ([76]; this is fixed in version 2.4). In another, you can create a `no-output` session where the kernel channels use the `splice` output mode (also fixed in version 2.4). In another, `lttng record` does not complain if there is no snapshot output parameter set defined when it is issued (for instance if you `lttng snapshot del-output` the default snapshot output parameter set) ([77]; this is fixed in version 2.4).

User-space events captured in per-process-ID mode (see `enable-channel`, B.4.8) fail to make it to persistent storage unless the `record` action is triggered *while the process is active*—this is not a bug, just a nasty feature [78]. Oddly, the snapshot sequence number is incremented by the successive `record` actions, even when they do not capture any data. User-space events can be captured in per-user-ID mode, *but only if no kernel events are being simultaneously captured* [79]. An empty tracing buffer (no events captured yet) will claim a successful snapshot record (“Snapshot recorded successfully for session <session>”) but nothing gets committed to storage, not even the session folder (nor `lttng-traces` if it did not exist yet) [80]. One would expect LTTng to create an empty trace instead. On the other hand, a session started without any event streams (no events enabled, no channels) will just fail to record snapshots.

B.4.14 start

Synopsis

```
lttng [<LTTNG_OPTIONS>] start [<SESSION_NAME>] [<OPTIONS>]
```

Description

Start tracing.

This command starts all of the session's tracers (there is currently no command option to start all sessions at once). If <SESSION_NAME> is omitted, the current session name is taken from the .lttngrc file if it exists (it won't exist if no session has been created so far or if the last command was `destroy`; this will result in an error).

Starting an already-started session yields a simple warning (“Tracing already started for session <session>”) but is otherwise harmless.

Options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` command option pre-empts the `list-options` command option if it appears first on the command line.

--list-options

Shows a simple listing of the `lttng start` options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` command option pre-empts the `help` command option if it appears first on the command line.

B.4.15 stop

Synopsis

```
lttng [<LTTNG_OPTIONS>] stop [<SESSION_NAME>] [<OPTIONS>]
```

Description

Stop tracing.

This command stops all of the session's tracers (there is currently no command option to stop all sessions at once). If <SESSION_NAME> is omitted, the current session name is taken from the .lttngrc file if it exists (it won't exist if no session has been created so far or if the last command was `destroy`; this will result in an error). Before returning, the command checks for data availability: it will wait until the trace buffers have all been flushed to storage (unless the session was running in snapshot mode, of course). Use the `no-wait` option to avoid this behaviour.

Stopping an already-stopped (or never-started) session yields a simple warning (“Tracing already stopped for session <session>”) but is otherwise harmless.

Options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` command option pre-empts the `list-options` command option if it appears first on the command line.

--list-options

Shows a simple listing of the `lttng stop` options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` command option pre-empts the `help` command option if it appears first on the command line.

-n, --no-wait

Don't wait for data availability. The command returns immediately, before the trace has finished committing itself to storage. This is rarely a problem from the command line, but could be important when using the API. By default, the command will remain busy until the buffers have been completely emptied to storage.

B.4.16 version

Synopsis

```
lttng [<LTTNG_OPTIONS>] version [<COMMAND_OPTIONS>]
```

Description

Show version information. Where the `lttng -v` command option is a one-liner, this command prints out a short text that, along with a few other things, explains the origin of the release's name. This command does not need nor launch the session daemon.

Options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` command option pre-empts the `list-options` command option if it appears first on the command line.

--list-options

Shows a simple listing of the `lttng` version options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` command option pre-empts the `help` command option if it appears first on the command line.

B.4.17 view

Synopsis

```
lttng [<LTTNG_OPTIONS>] view [<SESSION_NAME>] [<COMMAND_OPTIONS>]
```

Description

View a trace. By default, the `babeltrace` viewer will be used for text viewing. An option to invoke `lttv` will eventually be added. If `<SESSION_NAME>` is omitted, the current session name is taken from the `.lttngrc` file if it exists (it won't exist if no session has been created so far or if the last command was `destroy`; this will result in an error). Until live trace viewing is fully implemented, viewing the current session should only be done while it is stopped.

The command echoes the selected trace folder path and then launches the viewer. Once the viewer shuts down, the command passes its return value as its own. You may want to launch the command as a detached process and redirect its output (e.g. '`lttng view &> output &`').

This command does not launch the session daemon, even when asking for the session's full name.

Options

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` command option pre-empts the `list-options` command option if it appears first on the command line.

--list-options

Shows a simple listing of the `lttng view` options and then quits. When this option is specified, the remaining ones are ignored. The `list-options` command option pre-empts the `help` command option if it appears first on the command line.

-t, --trace-path <PATH>

Specifies the trace folder path for the viewer. Normally this is the path to the trace's topmost folder (e.g. `/home/<username>/lttng-traces/<session>-<date>-<time>`).

-e, --viewer <COMMAND>

Specifies the viewer and/or options to use. This will completely override the default viewer so make sure to specify a full command; if the command includes any spaces, it must be quoted. The trace folder path of the session (either the current session's or the one specified by the `trace-path` option) will be appended to the `<COMMAND>`.

Examples

In this first example, the trace is merely created before `lttng view` is invoked, so the trace folder does not even exist yet.

```
$ lttng create thesession
$ lttng view
Trace directory: /home/username/lttng-traces/thesession-20130917-092002

[error] Cannot open any trace for reading.

[error] opening trace "/home/username/lttng-traces/thesession-
20130917-092002" for reading.

[error] none of the specified trace paths could be opened.
```

Compare the output with `babeltrace`:

```
$ babeltrace ~/lttng-traces/thesession-20130917-092002

[error] Cannot open any trace for reading.

[error] opening trace "/home/username/lttng-traces/thesession-
20130917-092002" for reading.

[error] none of the specified trace paths could be opened.
```

The return value is 1 in both cases. To use `lttv` you call `lttv-gui`:

```
$ lttng view -e "lttv-gui --trace"
Trace directory: /home/username/lttng-traces/thesession-20130917-092002

** (lttv.real:25541): WARNING **: Cannot open directory
/home/username/lttng-traces/thesession-20130917-092002 (No such
file or directory)

** (lttv.real:25541): WARNING **: cannot open trace
/home/username/lttng-traces/thesession-20130917-092002
```

The return value is zero in this case. The 25541 quoted above is `lttv.real`'s process ID.

In this last example, the session daemon was killed and the `.lttngrc` file preserved:

```
$ lttn view
Error: Unable to list sessions. Session name thesession not found.
Is there a session daemon running?
Error: Command error
```

B.5 Exit values

On success 0 is returned; a positive value indicates an error. The `CMD_*` values are from `lttng-tools/src/bin/lttng/command.h`. Some commands apply to multiple objects (for instance `lttng enable-event -k --all`) and a single exit value is poorly suited to reflect the combination of successes and errors that may have occurred. The `stderr` stream is a better resource in such cases.

- | | |
|---|--|
| 0 | Success |
| 1 | Command error (missing options or incorrectly parameterised) |
| 2 | Undefined command (unrecognised command or option) |
| 3 | Fatal error |
| 4 | Command warning (minor error) |
| 5 | Unsupported command (unsupported combination of options) |

Examples:

- | | |
|---|---|
| 1 | Omitting a mandatory domain option |
| 2 | <code>lttng enable-event -k --all --output ...</code>
<code>lttng restart</code> |
| 3 | Out of memory |
| 4 | Enabling an already-enabled event |
| 5 | <code>lttng enable-event -k --all --loglevel TRACE_WARNING</code> |

Note that the semantics of the various failure modes are not defined systematically yet; for instance, trying to enable a filter on a kernel event should be *unsupported* but is reported as an *error*. This should straighten itself out as the code is steadily improved.

When using the LTTng API, an error code of 10 means success, and anything above that is an error. The `LTTNG_ERR_*` values are detailed in `lttng-tools/include/lttng/lttng-error.h` and you can use `lttng_strerror()` to get a human-readable string for the error code:

10	OK	21	URL already exists
11	Unknown error	22	Buffer type not supported
12	Undefined command	23	Session name not found
13	Session is running	24	Buffer type mismatch
14	Unknown tracing domain	25	Fatal error
15	Operation not supported	26	Not enough memory
16	No session found	27	Must select a session
17	Directory creation failed	28	Session name already exists
18	Error in session creation	29	No event found
19	No session daemon	30	Unable to connect to Unix socket
20	Error setting URL	31	Snapshot output set already exists

32	Permission denied	67	UST stop trace failed
33	Kernel tracer unavailable	68	64-bit UST consumer start failed
34	Kernel tracer incompatible with kernel version	69	32-bit UST consumer start failed
35	Kernel event already exists	70	UST create stream failed
36	Kernel session creation failed	74	UST listing events failed
37	Kernel channel already exists	75	UST event already exists
38	Kernel channel creation failed	76	UST event not found
39	Kernel channel not found	77	UST context already exists
40	Kernel channel disable failed	78	UST invalid context
41	Kernel channel enable failed	79	Tracing the kernel requires a root <code>lttng-sessiond</code> daemon
42	Kernel add context failed	80	Tracing already started
43	Kernel enable event failed	81	Tracing already stopped
44	Kernel disable event failed	82	Kernel event type not supported
45	Kernel open metadata failed	83	Non-default channel exists: channel name must be specified
46	Kernel start trace failed	84	UST tracer is not supported: rebuild <code>lttng-tools</code> to enable it
47	Kernel stop trace failed	97	Invalid parameter
48	Kernel consumer start failed	98	No UST consumer detected
49	Kernel create stream failed	99	No Kernel consumer detected
50	Session needs to be started once	100	Event already enabled at a different loglevel
51	Snapshot recording failed	101	Missing network data URL
53	Kernel listing events failed	102	Missing network control URL
54	UST calibration failed	103	Enabling consumer failed
55	UST event already enabled	104	<code>lttng-relayd</code> session creation failed
56	UST session creation failed	105	<code>lttng-relayd</code> not compatible
57	UST channel already exists	106	Invalid filter bytecode
58	UST channel creation failed	107	Out of memory for filter bytecode
59	UST channel not found	108	Filter already exists
60	UST channel disable failed	109	No consumer exists for the session
61	UST channel enable failed		
63	UST event enable failed		
64	UST event disable failed		
65	UST open metadata failed		
66	UST start trace failed		

B.6 Environment variables

Note that all command line options override environment variables.

B.6.1 `LTTNG_SESSIOND_PATH`

Allows you to specify the full session daemon binary path. You can use the `sessiond-path` program option to the same effect (see B.3). The `sessiond-path` program option pre-empts the `LTTNG_SESSIOND_PATH` environment variable.

This page intentionally left blank.

Annex C The `lttng-sessiond` program

C.1 Synopsis

```
lttng-sessiond [<OPTIONS>]
```

C.2 Description

The session daemon, acting as a tracing registry, allows you to interact with multiple tracers (kernel and user-space) inside the same container, a tracing session. Traces can be gathered from the kernel and/or instrumented applications, and committed to storage by several consumer daemons (see `lttng-ust(3)`). Aggregating those traces is done using a viewer, like the `babeltrace(1)` text viewer.

In order to trace the kernel, the session daemon needs to be running as root. LTTng provides the use of a tracing group (default: `tracing`). Whoever is in that group interacts with the root session daemon only and can thus trace the kernel (see Section 3.4). Session daemons can co-exist, meaning that you can have a session daemon running as `Alice` that can be used to trace her applications alongside with a root daemon or even a `Bob` daemon. Starting the session daemon at boot time is recommended for stable and long-term tracing. This is automated through the Upstart service by several installation packages.

The session daemon is in charge of managing trace data consumers by spawning them when the time has come. The user does not need to manage the consumer daemons (`lttng-consumerd`).

C.3 Program options

This program follows the usual GNU command line syntax with long options starting with two dashes [48] (optional use of the ‘=’ sign to separate long option names from their arguments is a GNU getopt extension).

-h, --help

Shows a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` option pre-empts the `version` option if it appears first on the command line.

-V, --version

Shows the version number (e.g. ‘`2.3.0`’) and then quits. When this option is specified, the remaining ones are ignored. The `version` option pre-empts the `help` option if it appears first on the command line.

-d, --daemonize

Starts as a daemon. By default it runs as a normal process. Oddly, trying to start a daemon again does not cause a warning to be issued, even though it fails (the command even returns error code zero).

-v, --verbose

Increases verbosity. There are three debugging levels (`v`, `vv`, and `vvv`); you can also repeat `verbose` up to three times (repeating `v` or `verbose` more than three times is the same as omitting the option entirely). The debug statements will print on `stderr`, so this option may not be combined with the `daemonize` option.

-Z, --verbose-consumer

Sets the verbose mode for the consumer daemons managed by the session manager. The consumer daemons have only one level of verbosity. They will forward their statements to the session manager, so `verbose-consumer` is only helpful if the session manager is not daemonized. The simplest approach to see the error message stream is to stop the session service and run `lttng-sessiond` in a shell.

-q, --quiet

Requests that `lttng-sessiond` produce no output at all.

-g, --group <GROUP_NAME>

Specifies the tracing group name (default: `tracing`). This is applicable only when starting an actual root daemon instance; if one is already running or if the command is starting a user-space daemon, the option is ignored. If the `<GROUP_NAME>` does not exist, you will get a warning message (“Warning: No tracing group detected”) and the tracing group will revert to “root”. About the only way to find out what the root session manager’s group setting is is to look up the group that `/var/run/lttng` belongs to (using an ‘`ls -dg`’ command).

Note that when a root daemon `lttng` command is rejected, the error message uses “tracing” instead of the daemon’s actual tracing group (e.g. “Tracing the kernel requires a root `lttng-sessiond` daemon or “tracing” group user membership”).

-S, --sig-parent

Requests that `lttng-sessiond` send `SIGCHLD` to its parent `pid` upon reaching readiness.

This is used by `lttng` to get notified when the session daemon is ready to accept commands. When building a third-party tool over `liblttng-ctl`, this option can be very handy to synchronise the control tool and the session daemon.

-N, --no-kernel

Prevents `lttng-sessiond` from providing kernel tracer support.

-p, --pidfile <FILE>

Write the process ID (PID) of the `lttng-sessiond` daemon to `<FILE>`. By default, `<FILE>` is `lttng-sessiond.pid`, in the `lttng-sessiond` run folder (`/run/lttng` for the root session daemon, `$HOME/.lttng` for a user-space session daemon; see Section 2.7). The consumer daemons use this file to contact their session daemon.

-c, --client-sock <PATH>

Specifies the `<PATH>` for the client-session daemon communication Unix socket (`client-lttng-sessiond`). The `lttng` client uses this socket to send commands to and receive responses from the session daemon. See Section 2.7 for the default locations.

-a, --apps-sock <PATH>

Specifies the `<PATH>` for the application registration Unix socket. User-space applications use this socket to register with the session daemon. See Section 2.7 for the default.

-C, --kconsumerd-cmd-sock <PATH>

Specifies the `<PATH>` for the kernel consumer daemon command Unix socket. See Section 2.7 for the default.

-E, --kconsumerd-err-sock <PATH>

Specifies the `<PATH>` for the kernel consumer daemon error Unix socket. See Section 2.7 for the default.

-u, --consumerd32-path <FILE>

Specifies the path for the 32-bit user-space consumer daemon binary.

-U, --consumerd32-libdir <PATH>

Specifies the `<PATH>` to the 32-bit user-space consumer daemon libraries.

-t, --consumerd64-path <FILE>

Specifies the path for the 64-bit user-space consumer daemon binary.

-T, --consumerd64-libdir <PATH>

Specifies the <PATH> to the 64-bit user-space consumer daemon libraries.

-G, --ustconsumerd32-cmd-sock <PATH>

Specifies the <PATH> for the 32-bit user-space consumer daemon command Unix socket. See Section 2.7 for the default.

-H, --ustconsumerd32-err-sock <PATH>

Specifies the <PATH> for the 32-bit user-space consumer daemon error Unix socket. See Section 2.7 for the default.

-D, --ustconsumerd64-cmd-sock <PATH>

Specifies the <PATH> for the 64-bit user-space consumer daemon command Unix socket. See Section 2.7 for the default.

-F, --ustconsumerd64-err-sock <PATH>

Specifies the <PATH> for the 64-bit user-space consumer daemon error Unix socket. See Section 2.7 for the default.

C.4 Environment variables

Note that all command line options will override environment variables.

C.4.1 LTTNG_CONSUMERD32_BIN

Specifies the 32-bit `lttng-consumerd` binary path. The `consumerd32-path` option overrides this variable.

C.4.2 LTTNG_CONSUMERD64_BIN

Specifies the 64-bit `lttng-consumerd` binary path. The `consumerd64-path` option overrides this variable.

C.4.3 LTTNG_CONSUMERD32_LIBDIR

Specifies the 32-bit library path to libconsumer.so. The consumerd32-libdir overrides this variable.

C.4.4 LTTNG_CONSUMERD64_LIBDIR

Specifies the 64-bit library path to libconsumer.so. The consumerd64-libdir overrides this variable.

C.4.5 LTTNG_DEBUG_NOCLONE

Debug-mode disabling use of clone/fork. When this environment variable is set, the session and consumer daemons won't use clone/fork when they create directories and files (the lttngr-traces directory, the trace folders, the trace files). The relay daemon is not affected. Insecure, but required to allow debuggers to work with lttngr-sessiond on some operating systems.

C.5 Limitations

For an unprivileged user running lttngr-sessiond, the maximum number of file descriptors per process is usually 1024. This limits the number of traceable applications since for each instrumented application there are two file descriptors per CPU and an additional socket for bidirectional communication.

For the root user, the limit is bumped up to 65535 file descriptors. A future version of LTtng will deal with this limitation.

This page intentionally left blank.

Annex D The `lttng-relayd` program

D.1 Synopsis

```
lttng-relayd [<OPTIONS>]
```

D.2 Description

The `lttng-relayd` daemon listens on the network and receives traces streamed by remote consumers. It does not require any particular permissions as long as it can write in the output folder and listen on the ports. It does not need or talk to the local session daemon: it runs completely on its own and converses only with remote session and consumer daemons.

Once a trace has been streamed completely, the trace can be processed by any tool that can process a local LTTng CTF trace. Once live streaming is completely implemented, a live trace being received by the daemon will be readable in the same way as a live local trace.

By default, `relayd` outputs the traces in `~/lttng-traces/<host-name>/<session-name>`.

The prefix (`~/lttng-traces`) can be configured on the `relayd` side (see below for the option), the other folders can be configured when creating the trace on the `sessiond` side.

The daemon listens on two ports, one for *control*, the other for *data*. The data stream port receives the tracing events themselves; everything else goes through the control port (metadata describing the trace, notification of sessions being created and destroyed/closed, etc.).

Trying to run the program a second time causes the following warning and fails, although the command nevertheless returns error code zero:

```
PERROR [14966/14969]: bind inet: Address already in use
(in lttcomm_bind_inet_sock() at inet.c:109)
```

D.2.1 Example

```
A$ sudo -H lttng create <session> -U net://<address>
B$ sudo -H lttng-relayd -C tcp://<address>:5342 -D tcp://<address>:5343
A$ sudo -H lttng enable-event ...
A$ sudo -H lttng start <session>
```

In this example, a session is created on machine A (the tracee) which directs its output to a network `<address>`. Machine B resides at this `<address>` and runs `lttng-relayd` to receive the trace. Note that B's `lttng-relayd` must be running (and reachable) for A's `lttng enable-event` to succeed.

D.3 Program options

This program follows the usual GNU command line syntax with long options starting with two dashes [48] (optional use of the ‘=’ sign to separate long option names from their arguments is a GNU getopt extension).

-h, --help

Show a summary of the possible options and commands and then quits. When this option is specified, the remaining ones are ignored. The `help` option pre-empts the `version` option if it appears first on the command line.

-V, --version

This option is no longer available. Show the version number and then quit. When this option is specified, the remaining ones are ignored. The `version` option pre-empts the `help` option if it appears first on the command line.

-d, --daemonize

Start as a daemon. By default it runs as a normal process. When running as a daemon, any error messages are lost; it may be much more convenient to “debug” any `lttng-relayd` commands in process mode before switching to daemon mode. Just like with `lttng-sessiond`, trying to start a daemon again fails but the command returns error code zero—the warning message is lost.

-v, --verbose

Increases verbosity. There are three debugging levels (`v`, `vv`, and `vvv`); you can also repeat `verbose` up to three times (repeating `v` or `verbose` more than three times is the same as omitting the option entirely). The debug statements will print on `stderr`, so this option may not be combined with the `daemonize` option.

-C, --control-port <URL>

Control stream port URL (`tcp://0.0.0.0:5342` is the default). The control stream port must be different from the data stream port but the address should otherwise be identical. Otherwise you get the following error:

```
PERROR: bind inet: Address already in use [in lttcomm_bind_inet_sock() at inet.c:89]
```

The default address of `tcp://0.0.0.0` should mean the daemon listens on all of the local machine’s IP addresses, at least for most Linux systems (on Windows systems, this address would fail). Currently, the daemon supports only the TCP protocol.

-D, --data-port <URL>

Data stream port URL (`tcp://0.0.0.0:5343` is the default). The data stream port must be different from the control stream port but the address should otherwise be identical.

-o, --output <PATH>

Output base folder (`~/lttng-traces` is the default). A `<PATH>` of length 3138 characters or less should be safe (excluding the trailing ‘/’ and null). This value is smaller than that quoted for the `lttng create` command’s `output` parameter (see Section B.4.3) because the latter includes the session name subdirectory.

D.4 Limitations

For now only TCP is supported on both the control and data stream ports. Control will always remain TCP-only since it is low-volume and needs to be absolutely reliable, but eventually the data connection could support UDP.

For an unprivileged user running `lttng-relayd`, the maximum number of file descriptors per process is usually 1024. This limits the number of connections and trace files that can be opened. This limit can be configured: see `ulimit(3)`.

This page intentionally left blank.

Annex E The `lttng-gen-tp` program

E.1 Synopsis

```
lttng-gen-tp [<OPTIONS>] <TEMPLATE_FILE>[ <TEMPLATE_FILE> ...]
```

E.2 Description

The `lttng-gen-tp` tool (a Python script) simplifies the generation of the user-space tracepoint files. It takes a simple template file (or several such files) and generates the necessary code to use the defined tracepoints in your application. The *Template file format* section (E.4) describes the contents of each template file, which boils down to the business part of a tracepoint header.

Currently, the tool can generate the `.h`, `.c` and `.o` associated with your user-space tracepoint provider. The generated `.h` can be directly included in your application. You can let the tool generate the `.o` or compile it yourself from the `.c`.

You can compile the `.c` into an `.o` or `.a` and link it with your application (see Section 4.1.3). To compile the resulting `.c` file, you need to add the options "`-L/usr/local/lib -ldl -lltng-ust`" as explained in Section 4.1. For example, to statically link the tracepoint provider `sample.tp` into an instrumented C++ application and run it, you would do:

```
$ lttng-gen-tp ./sample.tp
$ g++ -c -o cxx_app.o cxx_app.cxx
$ g++ -o cxx_app cxx_app.o sample.o -L/usr/local/lib -ldl -lltng-ust
$ ./cxx_app
```

`lttng-gen-tp` does not include the `#ifdef __cplusplus` guarded statements (see the *LTtng Quick Start Guide*, Section 4.4.1) in its generated `.h`, which may cause problems in some C++ contexts. You can easily fix this by wrapping your `.tp` with those guarded statements.

To compile your provider into a library (shared object `.so`), you will need to prefix the `lttng-gen-tp` invocation as shown below. This produces an `.o` that is compatible with shared object requirements:

```
$ CFLAGS="-fPIC" lttng-gen-tp ./sample.tp
$ gcc -I. -shared -Wl,-soname,libsample.so -Wl,-no-as-needed \
      -o libsample.so -L/usr/local/lib -ldl -lltng-ust sample.o
$ gcc -c -o dyn_app.o dyn_app.c
$ gcc -L/usr/local/lib -ldl -o dyn_app dyn_app.o
$ LD_PRELOAD=./libsample.so ./dyn_app
```

You will need to `#define TRACEPOINT_PROBE_DYNAMIC_LINKAGE` and `TRACEPOINT_DEFINE` in each instrumented application (`dyn_app.c`). The incantation used in the example above allows `dyn_app` to run with or without `libsample.so` in its environment. When running without, no tracepoint events are generated (see Section 4.1.3.2). Note also that the `.o` produced by `lttng-gen-tp` in this way will *not* be usable through static inclusion.

E.3 Program options

This program follows the usual GNU command line syntax with long options starting with two dashes [48] (optional use of the ‘=’ sign to separate long option names from their arguments is a GNU getopt extension).

-h, --help

Show a summary of the possible options and commands and then quit. When this option is specified, the remaining ones are ignored.

-v, --verbose

Turns verbosity on. This only means `lttng-gen-tp` will print out the command line used to generate the `.o` output file—if generated.

-o <OUTPUT_FILE>

Specify the generated file name. The type of the generated file depends on the file extension used (`.h`, `.c`, `.o`). This option can be specified multiple times to generate multiple file types at once; when doing so, only one input `<TEMPLATE_FILE>` may be specified. When no `<OUTPUT_FILE>` is specified, all three default files are generated, with the same base filename as the template file. The default files are: `.h`, `.c`, and `.o`. Whereas `.h` and `.c` can be generated independently, `.o` needs both `.h` and `.c` to be successful. The order in which the `<OUTPUT_FILE>`s appear is not important.

-a

This option is currently recognised but does nothing. It is internally named “`all`”.

E.4 Template file format

Each template file, which must have the usual extension `.tp`, contains a list of `TRACEPOINT_EVENT` definitions and other optional definition entries like `TRACEPOINT_LOGLEVEL` (see Section 4.1.2.2). `TRACEPOINT_EVENT_CLASS` and `TRACEPOINT_EVENT_INSTANCE` can also be used freely. You write them as you would write them in a C header file. You can add comments with `/* */`, `//` and `#`. The provider name (the first field of `TRACEPOINT_EVENT`) must be the same for the whole file.

E.4.1 Example

```
TRACEPOINT_EVENT(
    sample_tracepoint,
    message, // Comment
    TP_ARGS(char *, text),
```

```
/* Next are the fields */
TP_FIELDS(
ctf_string(message, text)
)
)
```

E.5 Environment variables

When the tool generates an .o file, it will look for the following environment variables.

E.5.1 CC

Specifies which C compiler to use. If the variable is not specified, the tool will try "cc" and then "gcc".

E.5.2 CFLAGS, CPPFLAGS, LDFLAGS

Flags passed directly to the compiler. Although LDFLAGS is not ignored by `lttng-gen-tp`, it doesn't have any effect because those flags only intervene at the linking stage, which `lttng-gen-tp` does not do.

This page intentionally left blank.

Annex F The babeltrace program

F.1 Synopsis

```
babeltrace [<OPTIONS>] <PATH> [<PATH2> ...]
```

F.2 Description

`babeltrace` is a trace viewer and converter reading and writing the Common Trace Format (CTF). Its main use is to pretty-print CTF traces into a human-readable text output sorted in strictly increasing time. The trace is read from the `<PATH>`, which is typically the trace's main folder. Multiple `<PATH>`s can be specified using separate arguments. A warning is issued for each `<PATH>` that could not be processed and an error occurs if no `<PATH>`s at all could be processed.

Roughly put, the `fields` and `no-delta` options (as well as the `verbose` option) choose *what* to print, the `names` and several `clock` options choose *how* to print it, and the `output` option chooses *where* to print it.

`babeltrace` has a plug-in architecture which makes it potentially expandable to other trace formats.

F.2.1 Common Trace Format (CTF) definitions

A CTF *Event Trace* is a time-ordered sequence of events. LTTng traces of multi-processor systems may contain simultaneous event records. Ideally, a disambiguation scheme should be used to predictably order simultaneous events. Since they perform have different origins, this typically means arbitrarily ordering the hosts, and then ordering the CPUs within each host using their IDs (a scheme like this can easily be extended to order other processors within a host, such as Graphics Processing Units (GPUs)). Until the 17 June 2013 6c9a50c commit, `babeltrace` did not do so, occurrences of truly simultaneous events being extremely rare. Since that commit, the event stream paths are used as secondary sort keys. Note that memory-mapped traces (see the `enable-channel` command's `output` option, B.4.8) have no stream paths, so the ordering of simultaneous events remains unspecified in their case.

A CTF *Event Stream* is an event trace subset, containing a subset of the trace event types. It corresponds to LTTng's notion of a *channel*. LTTng subdivides each channel into as many file streams as there are CPUs, but `babeltrace` merges them back into a single CTF event stream transparently. For the purposes of this document, a set of streams that differ only by CPU IDs is called a *stream bundle*. LTTng subdivides each channel into as many stream bundles as there are event sources: there will only be one stream bundle for the channel's kernel events, but there may be multiple stream bundles in user-space: one per process emitting events (a single process can migrate between CPUs and thus generate multiple file streams for each channel). Alternately, LTTng can map the stream bundles to the individual user-spaces: one per user-space (user ID) emitting events. Stream bundles that differ by originating process (or user) are also merged back into a single CTF event stream by `babeltrace`. In fact, `babeltrace` sorts its output by timestamps regardless of how many input <PATH>s are specified (but recall that timestamps may vary between traces for the same event, see Section 2.5).

A CTF *Event Packet* is a sequence of physically contiguous events within an event stream. This corresponds to LTTng's sub-buffers: as events occur, they are written out to a memory buffer which is subdivided into a number of sub-buffers, circularly linked to form a ring. CTF requires that events within a packet be in strictly increasing timestamp sequence, and this is the main reason why LTTng timestamps its event records with the time of slot reservation within each sub-buffer.

A CTF *Event* or *Trace Record* is the basic entry in a trace, called throughout this document an *event record*. A CTF *Event Identifier* relates to the class (type) of an event within an event stream. Event records are instances of the event classes (types).

CTF's TSDL (Trace Stream Description Language) is used to generate trace metadata which specifies the trace version, the event types available (including their fields, a.k.a. the event payloads), per-trace and per-stream event header descriptions, and per-stream and per-event event context descriptions. TSDL also describes the data types that make up event fields in minute detail (bit widths, byte-ordering, alignment, packing and padding, signedness, numeric bases, floating point representations, character encodings, etc.), including enumerations (mappings between integer types and tables of strings). TSDL allows all the classic compound types (structures, variants (tagged unions), arrays (statically sized), sequences (dynamically sized arrays), and strings (null-terminated)).

A full description of CTF is beyond the scope of this document. The specification of CTF is available on the Common Trace Format Web site, <http://www.efficios.com/fr/ctf/>.

F.2.2 babeltrace representation of CTF data

LTTng tends to restrict itself to integer and string types grouped in simple structures, although the stream event header includes a variant, and user-defined tracepoints can be fairly complex. In the `babeltrace` output, each value can be printed alone or labelled, depending on the choices of command options. Integers are usually represented in traditional decimal (e.g. “1234”, signed or unsigned) or hexadecimal (e.g. “0x4D2”) notation; network byte-ordering is also possible (e.g. a little-endian machine’s “0x1234” would be read as “0x3412”). Hexadecimal notation and network byte-ordering are specified by the payload metadata, not by `babeltrace` command options. Single, double or quadruple precision floating-point numbers are also possible (e.g. “1.23457e+308”). Strings are quoted (e.g. ““lttng-sessiond””) except for the ones in the event record header (the part up to and including the event name).

Statically-sized arrays are represented as a comma-separated series of values, enclosed by square brackets (e.g. “[-1, -2]”), with optional labels before each value and before the set (e.g. “longarray = [[0] = -1, [1] = -2]”). The array element labels consist of the square-bracketed array indices (e.g. “[0]”, “[1]”, etc.). Empty or singleton arrays are possible.

Sequences (dynamically-sized arrays) are represented as static arrays preceded by a length field named “`_<seqname>_length`” (e.g. “`_longsequence_length = 2, longsequence = [[0] = -1, [1] = -2]`” (labelled) or “`2, [-1, -2]`” (unlabelled)).

Structures are printed in “set notation”, as a comma-separated series of values enclosed by curly brackets (e.g. “{ 0, 1 }”) with optional labels before each value and before the set (e.g. “`stream.packet.context = { events_discarded = 0, cpu_id = 1 }`”). Before the 17 June 2013 5909f332 commit, a singleton or empty (field-less) structure was printed without its enclosing curly brackets when unlabelled (except for the `<PACKET_CONTEXT>` part (see below), probably because that part is not a true singleton, having extra fields revealed by the `verbose` option).

Enumerations are printed as a colon-separated string-integer value pair enclosed in parentheses (e.g. “(compact : 0)” or “(extended : 65535)”). Labelling adds a label to the enumeration itself and to the integer value (e.g. “`id = (compact : container = 0)`”). The integer’s label will always be ‘container’. Variants are a little more complex, because they consist of two parts which may be embedded within a great many different structures. The first part is the enumeration, which has already been described. The second part is the actual variant. When unlabelled, it appears as a simple two-level structure (e.g. “{ { 0, 196017830476081 } }” or “{ { 86548004 } }”). Before the 17 June 2013 5909f332 commit, a singleton or empty variant was printed without its inner enclosing curly brackets when unlabelled. Labelling acts as expected except that the inner structure’s label is the tag string (e.g. “`v = { extended = { id = 0, timestamp = 196017830476081 } }`” or “`v = { compact = { timestamp = 86548004 } }`”).

Nearly every part of the event record print line detailed below is optional (hence the brackets [] around each such part) because it may be absent from the CTF trace. The only exception is the event name. Note that the absence of a part from the trace is not necessarily possible with LTTng traces (as opposed to CTF-compliant traces produced by other means): for instance, LTTng traces include a timestamp with each record, without exception.

Here is an abstract record print line, first shown labelled (the fields in red currently can occur only in CTF traces from sources other than LTTng):

```
[ "timestamp = " <TIMESTAMP> ", " ]
[ "delta = " <TIME_DELTA> ", " ]
[ "trace = " <TRACE> ", " ]
[ "trace:hostname = " <HOST_NAME> ", " ]
[ "trace:domain = " <DOMAIN> ", " ]
[ "trace:procname = " <PROCNAME> ", " ]
[ "trace:vpid = " <VPID> ", " ]
[ "loglevel = " <LOGLEVEL> ", " ]
[ "model.emf.uri = " <EMF> ", " ]
[ "callsite = " <CALLSITES> ", " ]
"name = " <EVENT_NAME>
[ " " "stream.packet.context = { " <PACKET_CONTEXT> " }" ]
[ [", "] "stream.event.header = { " <EVENT_HEADER> " }" ]
[ [", "] "stream.event.context = { " <CONTEXT_STREAM> " }" ]
[ [", "] "event.context = { " <CONTEXT> " }" ]
[ [", "] "event.fields = { " <PAYLOAD> " }" ]
```

And here is the same line unlabelled:

```
[ "[" <TIMESTAMP> " ] "
[ "(" <TIME_DELTA> " ) "
[ <TRACE> " "]
[ <HOST_NAME> ]
[ <DOMAIN> ]
[ [":"] <PROCNAME> ]
[ [":"] <VPID> ]
[ [":"] <LOGLEVEL> ]
[ [":"] <EMF> ]
[ [":"] <CALLSITES> ]
[" "] <EVENT_NAME> ":" "
[ " { " <PACKET_CONTEXT> " }" ]
[ [", "] "{ " <EVENT_HEADER> " }" ]
[ [", "] "{ " <CONTEXT_STREAM> " }" ]
[ [", "] "{ " <CONTEXT> " }" ]
[ [", "] "{ " <PAYLOAD> " }" ]
```

Each part is detailed below.

TIMESTAMP

See the `clock-cycles` option for the various forms <TIMESTAMP> can take. If present in the trace, the <TIMESTAMP> is printed (you cannot choose to omit it). Labelling the <TIMESTAMP> is controlled by setting the `names` option to `header` or `all`.

TIME_DELTA

This has the form “+sec.ns” where ns is 9 digits wide. The very first record of each trace has a <TIME_DELTA> of “+?.?????????” (this is also true of any record not preceded by a timestamped record). This is omitted if the `no-delta` option is specified (before the 23 March 2013 e4497aa commit, a bug in `babeltrace` has the `no-delta` option ignored when the `fields` option is set to `all`). Labelling the <TIME_DELTA> is controlled by setting the `names` option to `header` or `all`.

TRACE

This is the stream bundle’s path, its *trace folder* (e.g. “/home/username/lttng-traces/thetrace/kernel”). Whereas the `babeltrace` input <PATH> may be the trace’s *session folder*, this part identifies each trace (leaf) folder separately because it is the smallest self-contained tracing data unit. Labelling the <TRACE> is controlled by setting the `names` option to `header` or `all`.

HOST_NAME

This is the trace’s host name (e.g. “edge-vb-u12”). This is handy when traces are streamed to remote storage, particularly if various instances of a distributed application are streamed simultaneously from multiple sources (hosts) to a central storage target. This is printed by default or if the `fields` option is set to `trace:hostname` or `all`. Labelling the <HOST_NAME> is controlled by setting the `names` option to `header` or `all`.

DOMAIN

This is the domain of the event record (e.g. “kernel” or “ust”). This is printed by default or if the `fields` option is set to `trace:domain` or `all`. Labelling the <DOMAIN> is controlled by setting the `names` option to `header` or `all`. Note that when they are unlabelled, the <HOST_NAME> and <DOMAIN> run into each other (they are *not* space- nor colon-separated).

PROCNAME

This is the record's originating process name, which exists only for per-process-ID user-space records (e.g. "sample"). This is printed by default or if the `fields` option is set to `trace:procname` or `all`. Labelling the `<PROCNAME>` is controlled by setting the `names` option to `header` or `all`. When unlabelled, the `<PROCNAME>` is separated from a preceding `<HOST_NAME>` and/or `<DOMAIN>` by a colon. Note that this is taken from the `name` field of `/proc/PID/status`, and is thus truncated to 15 characters.

VPID

This is the record's virtual process ID, which exists only for per-process-ID user-space records (e.g. "1256"). This is printed by default or if the `fields` option is set to `trace:vpid` or `all`. Labelling the `<VPID>` is controlled by setting the `names` option to `header` or `all`. When unlabelled, the `<VPID>` is preceded by a colon if preceded by at least one of the `<HOST_NAME>` through `<PROCNAME>` parts.

LOGLEVEL

This is the log level of the record (currently this exists only for user-space records). It has the form "`<LOGLEVEL_NAME> (<LOGLEVEL>)`" where the `<LOGLEVEL_NAME>` is a string like "TRACE_DEBUG" and the `<LOGLEVEL>` is its decimal value (e.g. "TRACE_WARNING (4)"). See Section 4.1.2.4 for the possible log level values. This is printed by default or if the `fields` option is set to `loglevel` or `all`. Labelling the `<LOGLEVEL>` is controlled by setting the `names` option to `header` or `all`. When unlabelled, the `<LOGLEVEL>` is preceded by a colon if preceded by at least one of the `<HOST_NAME>` through `<VPID>` parts.

EMF

This is the trace event's model URI, related to EMF (Eclipse Modeling Framework). It appears only in user-space event records. It has the form "`"<URI>"`" where the `<URI>` is a string like "`http://example.com/path_to_model?q=sample_component:message`". This is printed only if the `fields` option is set to `emf` or `all`. Labelling the `<EMF>` is controlled by setting the `names` option to `header` or `all`. When unlabelled, the `<EMF>` is preceded by a colon if preceded by at least one of the `<HOST_NAME>` through `<LOGLEVEL>` parts.

CALLSITES

This is callsite debug information. *Callsites* are the places in the instrumented code where tracepoints are invoked. LTTng currently has no capability to add callsites to trace event records; CTF-compliant traces from other sources may include this field, however. This block has the form “[<CALLSITE>, <CALLSITE>, …]” where the individual <CALLSITE> strings are like “<func>:<file>:<line>” (or “<func>@0x<address>:<file>:<line>” if the callsite IP (Instruction Pointer, a 64-bit unsigned integer) is specified), where <func> and <file> are strings, and <line> is a (decimal) line number. This is printed only if the fields option is set to callsite or all. Labelling the <CALLSITES> is controlled by setting the names option to header or all. When unlabelled, the <CALLSITES> are preceded by a colon if preceded by at least one of the <HOST_NAME> through <EMF> parts.

EVENT_NAME

This is the event’s name (e.g. “sched_switch” or “sample_component:message”); it matches the name reported by the lttng list -k -u command. User-space event names are of the form “<tracepoint_provider_name>:<event_name>”. This is *always* printed (the only such part). Labelling the <EVENT_NAME> is controlled by setting the names option to header or all. When unlabelled, the <EVENT_NAME> is preceded by a space if preceded by at least one of the <HOST_NAME> through <CALLSITES> parts.

PACKET_CONTEXT

This is the trace stream’s packet (sub-buffer) context. In CTF terms, this is actually two structures: the event stream header (a.k.a. event packet header) and event packet context. The stream packet context is normally a single-field object (a C struct) supplying the CPU ID, a very important datum. When the verbose option is specified, several other fields are added (timestamp_begin, timestamp_end, content_size, packet_size, events_discarded). Note that the content_size and packet_size are expressed in *bits*, not bytes. The <PACKET_CONTEXT> is always printed when available. Labelling the <PACKET_CONTEXT> itself is controlled by setting the names option to scope, while the labelling of its contents is controlled by setting the names option to context or all.

The <PACKET_CONTEXT> would be a natural place to find the event’s channel name. However, even though babeltrace unavoidably knows of the channel names, there is currently no way to get that datum to appear in its output.

EVENT_HEADER

This is the event's header (CTF event header). This is a variant, which `babeltrace` displays as an enumeration tag and its controlled field. This is printed only if the `verbose` option is specified. Labelling the `<EVENT_HEADER>` itself is controlled by setting the `names` option to `scope`, while the labelling of its contents is controlled by setting the `names` option to `context` or `all`. Examples (with and without labelling):

```
{ ( extended : 31 ), { { 0, 196017830476081 } } }
stream.event.header = { id = ( extended : container = 31 ),
    v = { extended = { id = 0, timestamp = 196017830476081 } } }

{ ( compact : 0 ), { { 196017830476081 } } }
stream.event.header = { id = ( compact : container = 0 ),
    v = { compact = { timestamp = 86548004 } } }
```

CONTEXT_STREAM

This is the event context supplied by the stream (CTF stream event context). This is another potentially multiple-field object, treated as the previous ones. It is added to a trace by the `lttng add-context` command (see B.4.1) and is always printed when available. Labelling the `<CONTEXT_STREAM>` itself is controlled by setting the `names` option to `scope`, while the labelling of its contents is controlled by setting the `names` option to `context` or `all`.

CONTEXT

This is the event context supplied by the event (CTF event context). This is another potentially multiple-field object, treated as the previous ones. This is always printed when available. Note that LTTng currently has no capability to add context at the event level [81]; `babeltrace` prints this only because CTF provides for it. Labelling the `<CONTEXT>` itself is controlled by setting the `names` option to `scope`, while the labelling of its contents is controlled by setting the `names` option to `context` or `all`.

PAYOUT

This is the event payload. This final object is potentially multiple-field, and treated as the previous ones. This is always printed when available. Labelling the `<PAYLOAD>` itself is controlled by setting the `names` option to `scope`, while the labelling of its contents is controlled by setting the `names` option to `payload` or `all`.

F.2.3 Examples

```
timestamp = 14:34:10.900109918,
delta = +?.??????????,
trace = /home/user/lttng-traces/session-20130306-143202/ust/pid/s-4179-20130306-143410,
trace:hostname = edge-vb-u12,
trace:domain = ust,
trace:procname = sample,
trace:vpid = 4179,
loglevel = TRACE_WARNING (4),
model.emf.uri = "http://example.com/path_to_model?q=sample:message",
name = sample:message,
stream.packet.context = {
    timestamp_begin = 666957554899173,
    timestamp_end = 666957633525228,
    content_size = 32744,
    packet_size = 32768,
    events_discarded = 0,
    cpu_id = 0
},
stream.event.header = {
    id = ( extended : container = 31 ),
    v = { extended = { id = 0, timestamp = 666957604091335 } }
},
event.fields = {
    message = "Hello World"
}

[14:34:10.900109918] (+?.??????????)
/home/user/lttng-traces/session-20130306-143202/ust/s-4179-20130306-143410
edge-vb-u12ust:sample:4179:TRACE_WARNING (4):
    "http://example.com/path_to_model?q=sample:message"
sample:message:
    { 666957554899173, 666957633525228, 32744, 32768, 0, 0 },
    { ( extended : 31 ), { { 0, 666957604091335 } } },
    { "Hello World" }

--names all:

timestamp = 11:02:08.064482311, delta = +0.000088385,
trace:hostname = edge-vb-u12, name = sys_geteuid,
stream.packet.context = { timestamp_begin = 4196192044293,
timestamp_end = 4196224429641, content_size = 2096768,
packet_size = 2097152, events_discarded = 0, cpu_id = 0 },
stream.event.header = { id = ( compact : container = 366 ), v = {
compact = { timestamp = 10140714 } } }, stream.event.context = {
ppid = 1, tid = 943, procname = "whoopsie", pid = 943,
perf_major_faults = 109 }, event.fields = { }
```

This names setting is simply the combination of the context (blue), header (yellow), payload (pink), and scope (green) settings.

F.3 Program options

This program follows the usual GNU command line syntax with long options starting with two dashes [48] (optional use of the '=' sign to separate long option names from their arguments is a GNU getopt extension).

-h, --help

Show a summary of the possible options and commands and then quit. When this option is specified, the remaining ones are ignored. The `help` option pre-empts the `list` option if it appears first on the command line.

-l, --list

List available input/output formats (`ctf`, `dummy`, `text`). When this option is specified, the remaining ones are ignored. The `list` option pre-empts the `help` option if it appears first on the command line.

-v, --verbose

Activate verbose mode. This option is also considered set when the `BABELTRACE_VERBOSE` environment variable is defined. The verbose mode prints descriptive statements (prefixed by '[verbose]') on `stderr` at the beginning of each trace processing, giving some of the `babeltrace` configuration and the trace's environment. A closing `stderr` comment concludes the output. More importantly, verbose output adds the `<EVENT_HEADER>` block to the output line and enriches the `<PACKET_CONTEXT>` block with several other fields.

Here is an example, highlighting the additional fields appearing with the verbose option:

```
timestamp = 15:33:38.911705190, delta = +0.000023455,
    trace = /lttng-traces/tys/kernel,
    trace:hostname = sds-dut-vb, trace:domain = kernel,
    name = sched_switch, stream.packet.context = {
        timestamp_begin = 105294547815883,
        timestamp_end = 105294738694877, content_size = 2096704,
        packet_size = 2097152, events_discarded = 0, cpu_id = 0 },
        stream.event.header = { id = { compact : container = 0 },
            v = { compact = { timestamp = 72271371 } } },
            event.fields = { prev_comm = "gnome-terminal",
                prev_tid = 3076, prev_prio = 20, prev_state = 1,
                next_comm = "cxx_app", next_tid = 7442, next_prio = 20 }
```

-d, --debug

Activate debug mode. This option is also considered set when the `BABELTRACE_DEBUG` environment variable is defined. The debug mode is meant for developers of babeltrace and adds a prodigious quantity of debugging statements (prefixed by ‘[debug]’) to the console output (the `<OUTPUT>` trace file is unchanged). Expect this debug stream, written to `stdout`, to be two or three times larger than the `<OUTPUT>`. A smaller set of messages goes to `stderr`, mostly tracking babeltrace’s decipherment of the trace’s metadata.

-i, --input-format <FORMAT>

Specify the input trace format. If the `<FORMAT>` is not specified, an error occurs. See the `list` option for the available formats. The default (and currently the only meaningful choice) is `ctf`. In earlier versions of babeltrace, specifying an `input-format` of `dummy` would cause a core dump [82]. This was fixed back in February of 2013.

-o, --output-format <FORMAT>

Specify the output trace format. If the `<FORMAT>` is not specified, an error occurs. See the `list` option for the available formats. The default is `text`. A `dummy` output format means babeltrace won’t output anything (except verbose messages on `stderr`). Currently, the `ctf` output format is *not* supported.

-w, --output <OUTPUT>

Specify the output trace file path. If the `<OUTPUT>` is not specified, an error occurs. The default is `stdout`. A bug in the Ubuntu package version prevents babeltrace from taking this option into account (this is fixed since the 14 February 2013 `c790c05` commit); the workaround (when outputting to `text`) is to redirect babeltrace’s `stdout` output to `<OUTPUT>`, like so:

```
$ babeltrace <PATH> [<PATH2> ...] [<OPTIONS>] &> <OUTPUT>
```

--no-delta

Do not print the time delta between consecutive events. This is the `(+<TIME_DELTA>)` part of the abstract format given above. This quantity can readily be calculated by comparing the two records’ timestamps; it is included by default for the human reader’s convenience.

Thus a line of output like:

```
[11:02:08.064482311] (+0.000088385) edge-vb-u12 sys_geteuid: [...]
```

Changes into:

```
[11:02:08.064482311] edge-vb-u12 sys_geteuid: [...]
```

A bug in the Ubuntu package version causes this option to be ignored if the `fields` option is set to `all`.

-f, --fields <NAME1>[,<NAME2>, ...]

Choose which fields to print. Roughly put, the `fields` and `no-delta` options (and to some extent the `verbose` option) allow you to choose *what* to print while the `names` and `clock` options allow you to choose *how* to print it.

If no `<NAME>` or an unrecognised `<NAME>` is specified, an error occurs. Each `<NAME>` can be one of: `all`, `trace`, `trace:hostname`, `trace:domain`, `trace:procname`, `trace:vpid`, `loglevel`, `emf`, `callsite`. Note that there is no `none`. Fields that are absent from the trace won't be printed at all. For instance, log levels are currently supported for the user-space only, so kernel events won't have any `loglevel` fields to display. Similarly, LTTng traces do not produce any `callsite` fields (yet).

Each `<NAME>` is a separate switch that can be turned on separately from the others (even `all`, it seems).

```
--fields all:  
[15:33:38.911705190] (+0.000023455)  
/home/daniel/lttng-traces/tys/kernel  
sds-dut-vbkernel  
sched_switch: { cpu_id = 0 }, { prev_comm = "gnome-terminal",  
    prev_tid = 3076, prev_prio = 20, prev_state = 1,  
    next_comm = "cxx_app", next_tid = 7442, next_prio = 20 }  
[15:33:38.911751794] (+0.000046604)  
/home/daniel/lttng-traces/tys/ust/pid/cxx_app-7442-20140313-153338  
sds-dut-vbus: cxx_app:7442:TRACE_WARNING (4)  
sample_component:event: { cpu_id = 0 },  
    { message = "CXXObject::message" }
```

This `fields` setting is simply the combination of the `trace` (blue), `trace:hostname` (yellow), `trace:domain` (pink), `trace:procname` (green, user-space only), `trace:vpid` (red, user-space only), `loglevel` (grey, user-space only), `emf` (absent), and `callsite` (absent) settings. The payloads of each event are labelled in the example because the default value of the `names` option is `payload,context`. The only context this output includes is the `<PACKET_CONTEXT>` ("`{ cpu_id = 0 }`").

Note: Setting the `fields` option to `all` overrides the `no_delta` option in the Ubuntu packages.

-n, --names <NAME1>[,<NAME2>, ...]

Specify how to label the data printed. Roughly put, the `fields` and `no-delta` options allow you to choose *what* to print while the `names` and `clock` options allow you to choose *how* to print it.

If no <NAME> or an unrecognised <NAME> is specified, an error occurs. Each <NAME> can be one of: none, all, (payload | args | arg), scope, header, (context | ctx). The default setting is ‘payload,context’. A choice of ‘all’ changes the record format to an explicit series of attribute-value pairs, for instance, while a choice of ‘none’ would list just the values (and be difficult to decipher unless one had knowledge of the trace configuration, in particular the <CONTEXT_STREAM>).

Each <NAME> is a separate switch that can be turned on separately from the others, except for none, which acts as master ‘off’ switch, and all, which acts as a master ‘on’ switch. Thus, any <NAME>s preceding none will be ignored, and any <NAME>s preceding or following all will be ignored (except a following none).

```
--names none:  
[11:02:08.064482311] (+0.000088385) edge-vb-u12  
sys_geteuid: { 4196192044293, 4196224429641, 2096768, 2097152, 0, 0 },  
{ ( compact : 366 ), { { 10140714 } } },  
{ 1, 943, "whoopsie", 943, 109 }, { }
```

In earlier versions of babeltrace, if the names option did not include payload and if the <PAYLOAD> had just one or zero fields, it was *not* enclosed in curly brackets ({}) [83]. This was fixed in June of 2013.

```
--names all:  
timestamp = 11:02:08.064482311, delta = +0.000088385,  
trace:hostname = edge-vb-u12,  
name = sys_geteuid,  
stream.packet.context = {  
    timestamp_begin = 4196192044293, timestamp_end = 4196224429641,  
    content_size = 2096768, packet_size = 2097152,  
    events_discarded = 0, cpu_id = 0 },  
stream.event.header = {  
    id = ( compact : container = 366 ),  
    v = { compact = { timestamp = 10140714 } } },  
stream.event.context = {  
    ppid = 1, tid = 943, procname = "whoopsie", pid = 943,  
    perf_major_faults = 109 },  
event.fields = {}
```

This names setting is simply the combination of the context (blue), header (yellow), payload (pink), and scope (green) settings.

--clock-cycles

Print the timestamps as [cycles]. When this option is set, the remaining clock options are ignored except for the `clock-force-correlate` option. The default is to print the timestamps as [hh:mm:ss.ns] expressed in babeltrace's local time (that is to say, the local time of the system running babeltrace, not the traced system's local time).

Here is the same timestamp expressed in different ways (the system running babeltrace is in the Eastern Standard Time zone, GMT-5):

```
(cycles)      [00000196017830476080]
(seconds)    [1361890904.423964840]
no_flags     [10:01:44.423964840]
(gmt)        [15:01:44.423964840]
(date)       [2013-02-26 10:01:44.423964840]
(gmt, date)  [2013-02-26 15:01:44.423964840]
```

--clock-offset <SECONDS>

Offset the clock by adding <SECONDS> seconds. If the <SECONDS> are not specified, an error occurs. <SECONDS> is an unsigned long integer. This option is ignored if the `clock-cycles` option is specified.

--clock-seconds

Print the timestamps as [sec.ns]. This is the time elapsed since 1970-01-01 00:00:00 UT, the Unix Epoch; as a consequence, the `clock-date` and `clock-gmt` options are ignored. The sec part is space-padded to a width of 3 digits if need be—this can only happen if the traced machine's clock is fried and the trace is captured within its first 17 minutes (1020 seconds) of up-time. The ns (nanoseconds) part is 9 digits wide. The `clock-cycles` option pre-empts this option.

--clock-date

Print the clock date and time as [yyyy-mm-dd hh:mm:ss.ns] instead of just [hh:mm:ss.ns]. The time and date are expressed in the current system's local time (which may be different from the traced machine's; currently the traced machine's local time reference is not stored in the trace). The `clock-cycles` option pre-empts `clock-seconds`, which pre-empts this option.

--clock-gmt

Use the GMT (more accurately Coordinated Universal Time (UTC)) time zone as the local one (instead of the system's local time zone). The `clock-cycles` option pre-empts `clock-seconds`, which pre-empts this option.

--clock-force-correlate

Assume that clocks are inherently correlated. This is meaningful only when multiple <PATH>s were specified to babeltrace (so it is printing several traces at once). This option does not affect how timestamps are rendered in the output.

F.4 Environment variables

Note that all command line options will override environmental variables.

F.4.1 BABELTRACE_VERBOSE

Activate verbose babeltrace output. The `verbose` program option has the same effect.

F.4.2 BABELTRACE_DEBUG

Activate debug babeltrace output. The `debug` program option has the same effect.

This page intentionally left blank.

Annex G The babeltrace-log program

G.1 Synopsis

```
babeltrace-log [<OPTIONS>] <OUTPUT>
```

G.2 Description

Convert a text log (read from standard input) to CTF and write it out to the <OUTPUT> folder.

This program should be considered a very rough draft. In particular, it cannot read babeltrace output to turn it back into a trace folder. It merely wraps each line of the input log into an event named `string` that has a payload consisting in a single character string field named `str`. These events don't even have a timestamp unless the `-t` option (see below) is used.

If multiple <OUTPUT> values are specified, only the last one is used. The <OUTPUT> folder will be created and must *not* already exist, otherwise you get a ‘`mkdir: File exists`’ error. Normally the program’s input is piped in, as in this example:

```
$ dmesg | babeltrace-log -t ~/lttng-traces/dmesg-trace
```

You could also use a here-document, a file, etc.

G.3 Program options

Note that this program, unlike the others, has no long-named options.

-h

Show a summary of the possible options and then quit. When this option is specified, the remaining ones are ignored. The `help` option is currently *not* recognised.

-t

Expect each input line to begin with a timestamp in either the `[sec.\usec]` or the `[yyyy-mm-dd hh:mm:ss.\usec]` format. The timestamps are assumed to represent time elapsed since 1970-01-01 00:00:00 UT (the Unix Epoch) for the `[sec.\usec]` format, or to be expressed in the GMT time zone for the `[yyyy-mm-dd hh:mm:ss.\usec]` format. Note that `dmesg`, for instance, actually records the time elapsed since machine start-up.

This page intentionally left blank.

Annex H The lttngtop program

H.1 Synopsis

```
lttngtop <TRACE_PATH> [<OPTIONS>]
```

H.2 Description

`lttngtop` is an ncurses interface for reading and browsing traces recorded by the LTTng tracer; it displays various statistics—as of now, the CPU usage, per file/process I/O bandwidth and `perf` counters. The current version only supports offline traces (traces that have been completed), but a live version (able to browse traces while they’re being generated) should be available in the near future. `lttngtop` is not, by far, a replacement for `babeltrace` or any other proper trace viewer: it is meant only to allow one to browse activity statistics over a sliding one-second window. In particular, it does not display trace events at all.

To work correctly, `lttngtop` requires that the `pid`, `procname`, `tid` and `ppid` context information be included in the trace. This is done with the `add-context` command (see B.4.1 and the example below).

If you want the CPU activity view (`CPUtop`), you need the `sched_switch` kernel event. If you want the I/O statistics (`IOTop`), you need to enable system call tracing. If you want the performance counters (`PerfTop`), you need to enable them for the events you are interested in (or all of them). Note that the hardware limits the number of performance counter you can enable: check `dmesg` for the details (see B.4.1). The other two events `lttngtop` explicitly looks for are `sched_process_free` and `lttng_statedump_process_state`. The `IOTop` view relies on the system calls `sys_write`, `sys_read`, `sys_open`, and `sys_close`, as well as the LTTng events `exit_syscall` and `lttng_statedump_file_descriptor`.

The following example creates a trace with all kernel events enabled, the mandatory context information and three performance counters. Kernel tracing requires running as root or membership in the `tracing` group and a running `lttng-sessiond` root service. You can use whatever you like for the session name:

```
$ lttng create <session>
$ lttng enable-event -k -a
$ lttng add-context -k -t pid -t procname -t tid -t ppid \
    -t perf:cache-misses -t perf:major-faults \
    -t perf:branch-load-misses
$ lttng start
$ sleep 10
$ lttng stop
$ lttng destroy
$ lttngtop ~/lttng-traces/<session>-<date>-<time>
```

`lttngtop` starts in CPU Top mode and displays partial CPU usage rankings while scanning the trace (see Figure 51 below). The Status window at the bottom displays Starting display while the scan is in progress, but unfortunately does not display any particular message once the scan is complete, so you will need to watch for quiescence of the display.

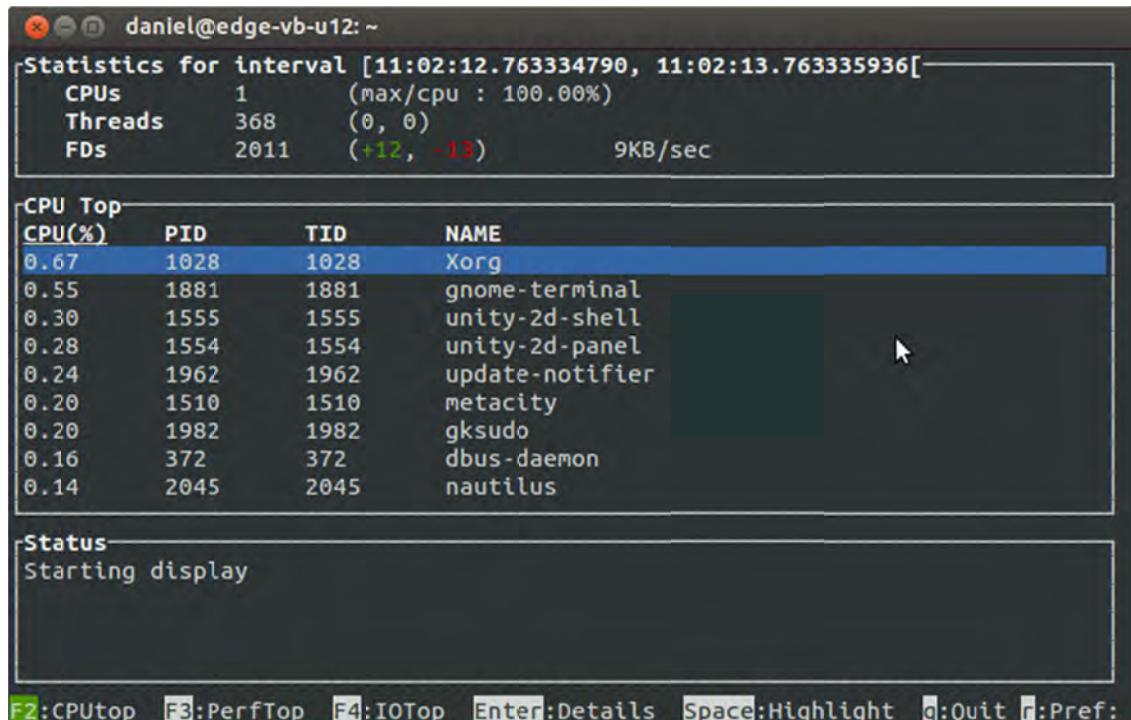


Figure 51: The starting lttngtop window.

The time interval upper bound (end point) of the `lttngtop` display is the timestamp of the first event to lie one second or more after the interval's lower bound (start point).

The Threads line displays the number of threads extant during the interval, followed by the number of threads launched and the number of threads terminated during the interval. For instance, “368 (+1, -2)” would mean 368 threads were active during the interval, one of which was created during the interval, and two of which were terminated.

Similarly, the FDs line tracks the number of file descriptors (network or local) and the total throughput for the interval. In Figure 51 above, 2011 file descriptors were active, including 12 new ones (created during the interval) and 13 old ones (closed during the interval), for a total throughput of 9 kilobytes per second. Despite using ‘K’ instead of ‘k’, `lttngtop`’s unit prefixes are all decimal: K is 1000, M is 1000 K, G is 1000 M. This is in contrast with the `lttng` usage for memory sizes in Section B.4.8 (`subbuf-size` and `tracefile-size`), and Section B.4.13 (`max-size`).

`lttngtop` will show a very large spike in the Threads and FDs lines at a trace’s beginning. This is an artefact of the missing data for the interval preceding the first one.

The listing of processes of the various displays (CPUTop, PerfTop, IOTop) is scrolling, but there is no indication of the current position within the overall list; this is a limitation of ncurses.

H.2.1 Key bindings

q: Quit

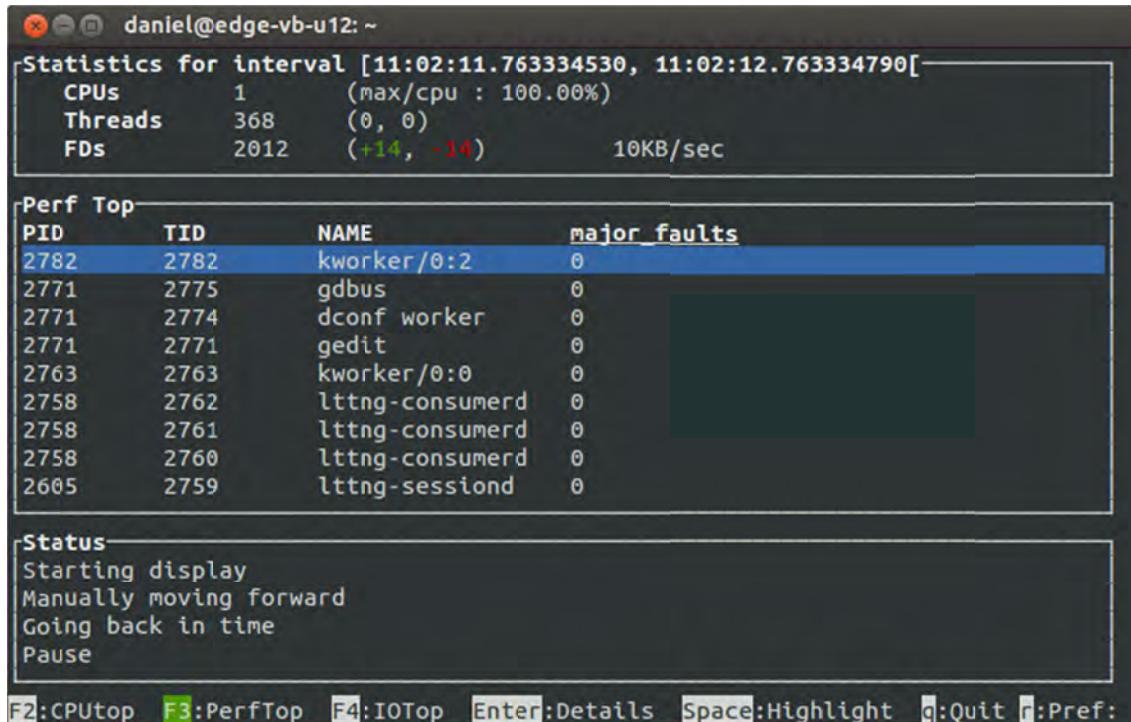
Exit the program.

F2: CPUTop

Switch to the CPU Top view (middle window) which displays the CPU usage of each process over the trace's interval. This is the starting display.

F3: PerfTop

Switch to the PerfTop view (middle window) which displays the performance counters (PMU) value of each process (if enabled during tracing).



The screenshot shows a terminal window titled "daniel@edge-vb-u12: ~". It displays three windows side-by-side. The left window is a status bar showing "Statistics for interval [11:02:11.763334530, 11:02:12.763334790[". The middle window is titled "Perf Top" and lists processes with their major faults. The right window is a status bar with controls for moving forward and backward in time, pausing, and exiting.

PID	TID	NAME	major_faults
2782	2782	kworker/0:2	0
2771	2775	gdbus	0
2771	2774	dconf worker	0
2771	2771	gedit	0
2763	2763	kworker/0:0	0
2758	2762	lttng-consumerd	0
2758	2761	lttng-consumerd	0
2758	2760	lttng-consumerd	0
2605	2759	lttng-sessiond	0

Status
Starting display
Manually moving forward
Going back in time
Pause

F2:CPUTop F3:PerfTop F4:IOTop Enter:Details Space:Highlight q:Quit r:Pref:

Figure 52: The PerfTop display.
Here only one perf counter was available (`major_faults`).

F4: IOTop

Switch to the `IOTop` view (middle window) which displays the I/O usage of each process. As of now this consists of three columns: the number of bytes read during the one-second interval currently chosen (R), the number of bytes written during that same interval (W), and the total number of bytes read from and written so far (from the trace's start until the end of the current interval) (Total).

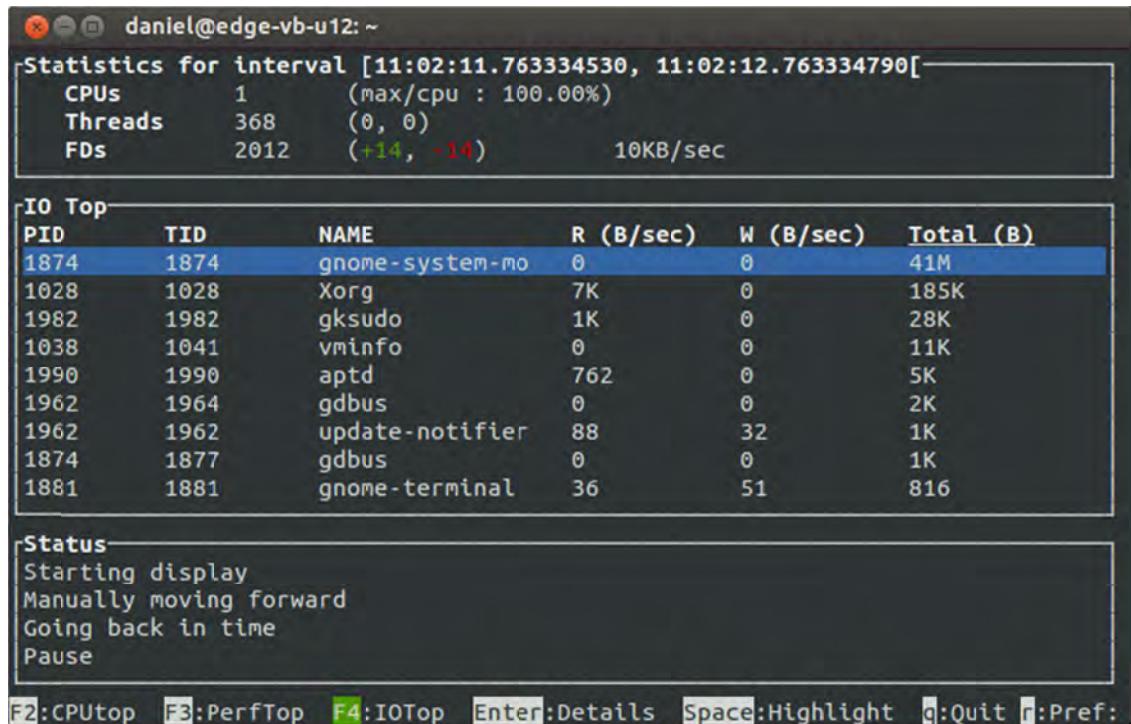


Figure 53: The IOTop display.

Right arrow: Move forward in time

Display the next second of data. You do not have to wait for the display to refresh before hitting the arrow key again. Also works while in the preferences display.

Left arrow: Move backward in time

Display the previous second of data, automatically switch to pause if not already enabled. Also works while in the preferences display.

Up arrow / k: Move UP the cursor

Move up the blue line to select the previous process (or the previous field while in the preferences display). This works even in details display, although the focus of the details display doesn't change; you will need to toggle it off and on to resynchronise with your underlying list focus.

Down arrow / j: Move DOWN the cursor

Move down the blue line to select the next process (or the next field while in the preferences display). This works even in details display (see above).

Space: Highlight

Toggle highlight of the process under the blue line. You can highlight as many processes as you like. They remain highlighted as you switch views. You can also toggle the highlight of a process while in its details view. Ignored while in the preferences display.

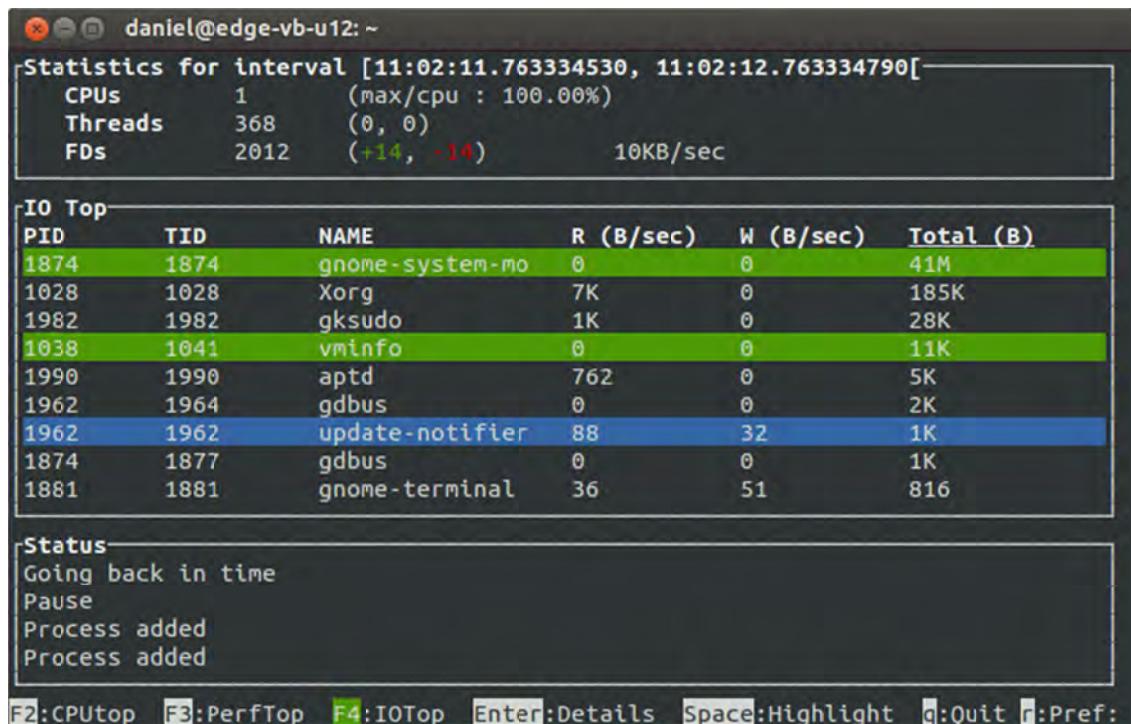


Figure 54: An IOTop display with two processes highlighted (in green).

Enter: Process details

Toggle the display of the selected process's details. Ignored while in the preferences display.

The screenshot shows a terminal window titled 'daniel@edge-vb-u12: ~'. It displays performance statistics for an interval from 11:01:57.759331393 to 11:01:58.759331539. The statistics include:

- CPU usage: 1 CPU at 100.00% (max/cpu)
- Threads: 368 (0, 0)
- FDs: 2009 (+15, -13) at 17KB/sec

Process details

Name	vminfo
TID	1041
PID	1038
PPID	1
CPU	0.00 %
READ B/s	3K
WRITE B/s	0
perf_major_faults	0

FD READ WRITE FILENAME

Status

- Cannot rewind, last data is already displayed
- Cannot rewind, last data is already displayed
- Cannot rewind, last data is already displayed
- Going forward in time

F2:CPUTop F3:PerfTop F4:IOTop Enter:Details Space:Highlight q:Quit r:Pref:

*Figure 55: A process details display.
The I/O details are for the current interval.*

t: Threads

Toggle threads display on and off. **This toggle isn't shown in the interface.** By default, both parent and child processes are shown. By toggling thread display off, you get the parents only. While the threads are toggled off, lttngtop may appear occasionally sluggish as you scroll up and down the process list. This is because the cursor is still visiting the hidden thread lines.

r: Preferences

Toggle display of the preference menu for the current view. This menu allows selection of the columns displayed (only in PerfTop view for now) and the column to sort on (always in descending order). The Details view offers a sort on FD, Read, or Write, but this seems to have no effect. Use 's' to choose the column to sort on, and 'space' to toggle the column to display. Optional columns toggle between '[]' and '[x]', while fields marked '[-]' cannot be toggled. You can also use the 'Escape' key to close the preferences menu.

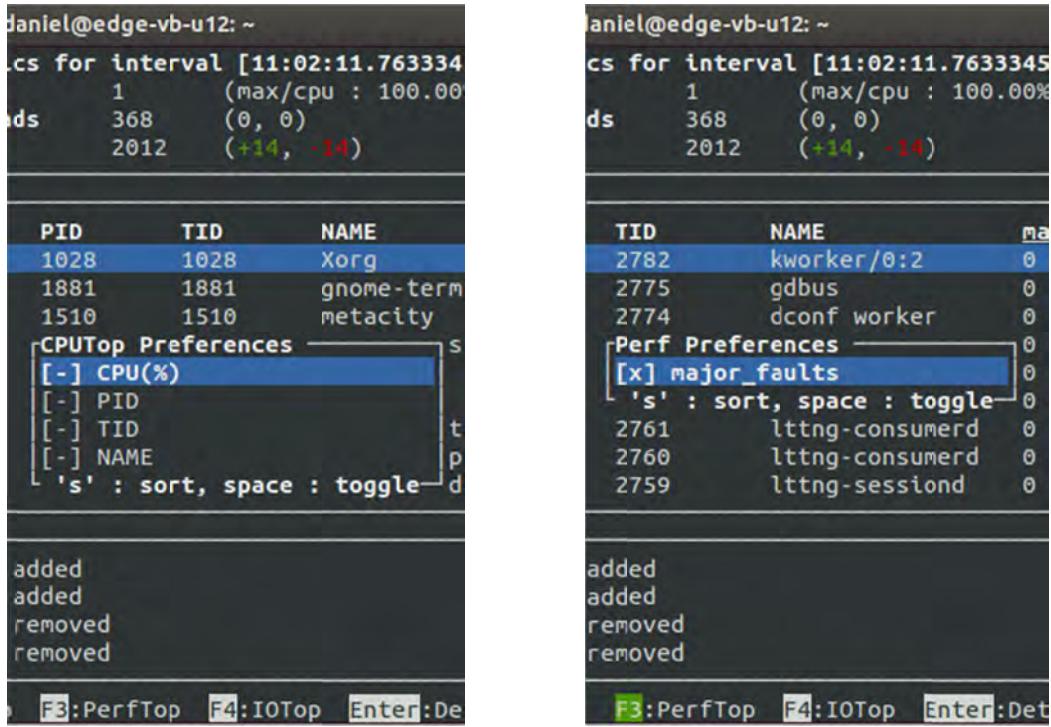


Figure 56: Two preferences displays.

On the left, you cannot toggle the display of the fields, just the sort key. On the right, you can do both (although there is but one field).

s: Sort

In a preference menu, sort (in descending order) on the currently selected field.

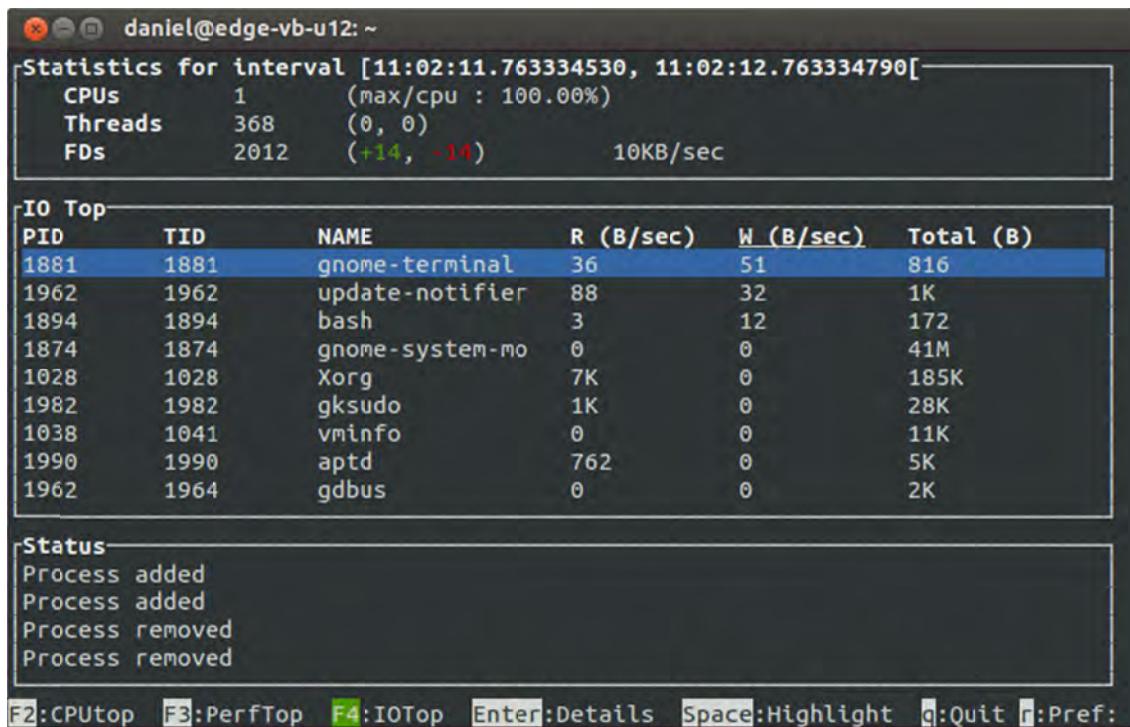


Figure 57: An IOTop display sorted on bytes written (W).

Note how the sort column's header is underlined.

>: Sort the next column

Sort using the next column. Ignored while in the preferences display. In the Details display, it moves the focus between FD, READ, and WRITE at the bottom of the details display but this has no effect (this may be an artefact of an as-yet-to-be-implemented feature of lttngtop).

<: Sort the previous column

Sort using the previous column. Also works in the details display (see above).

p: Pause/Resume

Pause the display. Hit 'p' again to resume the refresh. This only works while the trace has not finished scanning.

H.3 Program options

-h, --help

Show a very brief synopsis of command usage and then quit.

Annex I The lttng-ust library

I.1 Description

LTTng-UST, the Linux Trace Toolkit Next Generation UserSpace Tracer, is a port of the low-overhead tracing capabilities of the LTTng kernel tracer to user-space. The library `liblttng-ust.so` enables tracing of applications and libraries.

It is found in the `lttng-ust` package. An older version is in the `liblttng-ust-dev` Ubuntu package.

I.2 Environment variables

I.2.1 `LTTNG_UST_DEBUG`

Setting this variable activates `liblttng-ust` debug output (for the daemons that use it). It is also possible to turn debug output on *permanently* while making `liblttng-ust` (see Section 3.3.3.3) by adding `CPPFLAGS` to the `make` line:

```
$ make CPPFLAGS="-D LTTNG_UST_DEBUG" &> make.log
```

I.2.2 `LTTNG_UST_REGISTER_TIMEOUT`

This environment variable can be used to specify how long the applications should wait for the `sessiond "registration done"` command before proceeding to execution of the main program. The time-out value is specified in milliseconds and the default is 3000 ms (3 seconds). The value 0 means "don't wait". The value -1 means "wait forever". Setting this environment variable to 0 is recommended for applications with time constraints on the process start-up time.

This page intentionally left blank.

Annex J The `lttng-ust-cyg-profile` libraries

J.1 Synopsis

This extension of the `lttng-ust` library provides a simple means of instrumenting applications with the equivalent of the kernel's `syscall` facility. This can be done by itself, or in conjunction with user-defined tracepoint instrumentation.

J.2 Description

For compilers that provide the code generation option `finstrument-functions` (such as LLVM/CLang and GCC; e.g. ‘`gcc -finstrument-functions`’), the `lttng-ust` package provides shared libraries (`liblttng-ust-cyg-profile.so` and `liblttng-ust-cyg-profile-fast.so`) that allow users to trace the function flow of their applications. Function tracing comes in two flavours, fast or verbose, each providing different trade-offs. The flavour to use is chosen when launching the instrumented application.

To instrument an application for function tracing, it is simply compiled with the `finstrument-functions` option. There is no need to modify the source code at all (though the source code itself must be available). It might be necessary to limit the number of source files where this option is used to prevent excessive amounts of trace data from being generated at run-time. Usually there are additional compiler flags that allow the specification of a selection of function instrumentations. For example, with `gcc` the `no_instrument_function` function attribute can be used within the code, or the `finstrument-functions-exclude-function-list` and/or `finstrument-functions-exclude-file-list` options can be added during compilation. All of these serve to indicate which function calls should be excluded from tracing.

The `finstrument-functions` option instruments functions by adding function hooks right after each function entry and just before each function exit. These hooks are inert by default. When one of the provided shared libraries (`liblttng-ust-cyg-profile.so` for verbose tracing or `liblttng-ust-cyg-profile-fast.so` for fast tracing) is preloaded (`LD_PRELOAD`), the profiling hooks emit LTTng events.

Using this feature can result in massive amounts of trace data being generated by the instrumented application if there is a lot of function calling going on. Application run-time can also be considerably affected. Be careful on systems with limited resources.

Fast profiling

The library `liblttng-ust-cyg-profile-fast.so` is a lightweight variant that should only be used when it can be guaranteed that the complete event stream will be recorded without any missing events. Any kind of duplicate information is left out. If any events do go missing, trace analysis applications won't be able to accurately reconstruct the application state as a function of time downstream of the missing events.

Each function entry is recorded as `lttng_ust_cyg_profile_fast:func_entry` with a single payload field `addr` holding the address of the called function. Each function exit is recorded as `lttng_ust_cyg_profile_fast:func_exit`, with an empty payload. Both events are loglevel `TRACE_DEBUG_FUNCTION(12)`.

Verbose profiling

This is a more robust variant which also works when events might be discarded (or omitted from application startup). Trace analyzers may still need extra information to be able to reconstruct the program flow, depending on how many of the events are missing.

The function entry and exit events have the same names as in fast profiling except for the provider part (`lttng_ust_cyg_profile:func_entry` and `lttng_ust_cyg_profile:func_exit`), and they both have a payload containing two fields: `addr` (address of the called function) and `call_site` (address at which the call occurred). Matching `func_entry` and `func_exit` pairs will have matching `addr` and `call_site` payloads.

Example

Consider this simple application:

```
#include <stdio.h>
#include <unistd.h>
int NLOOPS = 10000;

int f(int ceiling, int i)
{
    return ceiling - i;
}

int main(int argc, char **argv)
{
    int i = 0;
    char themessage[20]; //Can hold up to "Hello World 9999999\0"

    fprintf(stderr, "sample starting\n");
    for (i = 0; i < NLOOPS; i++) {

        sprintf(themessage, "Hello World %u", f(NLOOPS, i));
        usleep(1);
    }
    fprintf(stderr, "sample done\n");
    return 0;
}
```

This can be compiled and linked using these calls:

```
$ gcc -c -o uninstr.o sample_uninstr.c -finstrument-functions  
$ gcc -o sample_uninstr uninstr.o -L/usr/local/lib
```

And it can be run like so:

```
$ LD_PRELOAD=/usr/local/lib/liblttng-ust-cyg-profile.so  
./sample_uninstr
```

The events can be captured by enabling the appropriate user-space events in the trace. For instance:

```
$ lttn create cyg  
$ lttn enable-event -u -a --loglevel-only TRACE_DEBUG_FUNCTION  
$ lttn start  
[run the application]  
$ lttn destroy
```

This page intentionally left blank.

Annex K The `lttng-health-check` function

K.1 Synopsis

```
#include <lttng/lttng.h>
int lttng_health_check(enum lttng_health_component c);
Link with -llttng-ctl
```

K.2 Description

The `lttng_health_check` function is used to check the session daemon health for either a specific component `c` or for all of them. Each component represents a subsystem of the session daemon. Those components are set with health counters that are atomically incremented once reached. An even value indicates progress in the execution of the component. An odd value means that the code has entered a blocking state which is not a `poll` wait period.

Bad health is defined by a fatal error code path reached or any IPC (Inter-Process Communication) used in the session daemon that was blocked for more than 20 seconds (the default timeout). The condition for this bad health to be detected is that one or more of the counters are odd.

The health check mechanism of the session daemon can only be reached through the health socket which is separate from the other sockets. An isolated thread serves this socket and only computes the health counters across the code when asked by the `lttng` control library (using this call). This subsystem is highly unlikely to fail due to its simplicity. Should it fail nevertheless, it has no consequences for tracing except that there is no longer any way of checking the health of the various LTTng components.

The `c` argument can be one of the following values:

K.2.1 `LTTNG_HEALTH_CMD`

The function will report the health of the command subsystem (see the `client-sock` option of `lttng-sessiond`, Section C.3).

The command subsystem is an `lttng-sessiond` thread which handles user commands coming from the `liblttng-ctl` API or the `lttng` command-line interface. It dispatches the commands to the tracepoint providers and consumer daemons, and sends the results to the client.

If this component malfunctions, LTTng is no longer responsive to commands though it may keep on capturing trace data. The session daemon will likely need to be restarted and the sessions recreated from scratch.

K.2.2 LTTNG_HEALTH_KERNEL

The function will report the health of the kernel tracer streams and the main channel of communication with the kernel tracer modules (/proc/lttng).

The root session daemon uses a thread to monitor all kernel channels of all sessions. The only action this thread takes is when a new CPU is detected (hot-plugging): the thread then notifies the kernel consumer of the happenstance.

If this component malfunctions, the kernel tracer is not usable anymore by LTTng.

K.2.3 LTTNG_HEALTH_CONSUMER

The function will report the health of the consumer daemons (see the `kconsumer*` and `ustconsumer*` options of `lttng-sessiond`, Section C.3). A session daemon can spawn up to three consumer daemons (one for the kernel domain, and a pair of 32- and 64-bit daemons for user-space).

The session daemon's consumer management threads (up to three, one for each consumer daemon) merely handle UST metadata requests and the consumer daemon exit code (when it fails during tracing).

If any one of the daemons malfunction, the corresponding tracing is likely no longer being committed to storage, although the events may still be captured to the memory buffers.

K.2.4 LTTNG_HEALTH_APP_REG

The function will report the health of the application registration socket (see the `apps-sock` option of `lttng-sessiond`, Section C.3). Upon startup, applications instrumented with `lttng-ust` try to register with the session daemon through this socket. If the session daemon isn't started yet, an application thread tries again every five seconds.

The application registration socket is serviced by a session daemon thread which accepts a pair of messages from each application (these messages describe the application's command and notification sockets) and stores those in a queue. This queue is watched by the next session daemon thread, the application registration dispatcher (see `LTTNG_HEALTH_APP_REG_DISPATCH`, below).

If this component malfunctions, the user-space tracer is no longer reliable as far as new applications or processes are concerned.

K.2.5 LTTNG_HEALTH_APP_REG_DISPATCH

This member of the `lttng-health-component` enum is currently undocumented (even as of version 2.4).

The function will report the health of the application registration dispatcher.

This session daemon thread processes the command and notification sockets registered by applications through the application registration socket (see `LTTNG_HEALTH_APP_REG`, above). The command socket is sent to the `LTTNG_HEALTH_APP_MANAGE` thread, and the notify socket is sent to the `LTTNG_HEALTH_APP_MANAGE_NOTIFY` thread.

If this component malfunctions, the user-space tracer is no longer reliable as far as new applications or processes are concerned.

K.2.6 `LTTNG_HEALTH_APP_MANAGE_NOTIFY`

This member of the `lttng-health-component` enum is currently undocumented (even as of version 2.4).

The function will report the health of the application notification socket manager subsystem.

This session daemon thread receives notifications from applications about available events and context fields (associated with channels; see the `lttng add-context` command, Section B.4.1).

If this component malfunctions, the user-space tracer is no longer reliable as far as new events or channel contexts are concerned.

K.2.7 `LTTNG_HEALTH_APP_MANAGE`

The function will report the health of the application command socket manager subsystem.

This session daemon thread watches the application command sockets; their closure indicates application shutdown (more accurately, shutdown of the application's tracepoint provider) and triggers unregistration.

If this component malfunctions, closure of application trace files (if tracing in per-process ID mode) will be delayed until the session is destroyed. It is unlikely any events will be lost.

K.2.8 `LTTNG_HEALTH_HT_CLEANUP`

This member of the `lttng-health-component` enum is currently undocumented (even as of version 2.4).

The function will report the health of the health table clean-up thread.

This is a minor thread whose only function is to be rejoined by the `lttng-sessiond` daemon's main thread during its shutdown in order to release the health management data structures. It plays no role during tracing.

If this component malfunctions, there may be some memory leakage once the session daemon shuts down. Traces are not affected.

K.2.9 LTTNG_HEALTH_ALL

The function will report the combined health of all components. It simply ANDs together the results of the other interrogations, meaning it will return OK only if each component is OK.

K.3 Return value

The function returns 0 if the health is OK, 1 if it's not. A return code of -1 indicates that the control library was not able to connect to the session daemon health socket.

K.4 Limitations

For the LTTNG_HEALTH_CONSUMER component, you cannot know which consumer daemon has failed but only that either the consumer subsystem as a whole has failed, or that one of the `lttng-consumerd` daemons has died.

List of symbols/abbreviations/acronyms/initialisms

API	Application Programming Interface
APT, apt	Advanced Packaging Tool
ARM	Advanced RISC Machine
ASCII	American Standard Code for Information Interchange
BSD	Berkeley Software Distribution
CA	California
CPU	Central Processing Unit
CTF	Common Trace Format
DND	Department of National Defence
DRDC	Defence Research and Development Canada
DREnet	Defence Research Establishment network
DSTKIM	Director Science and Technology Knowledge and Information Management
DVD	Digital Versatile Disc
DVD-ROM	DVD – Read-Only Memory
EAX	Extended Accumulator X (general-purpose) ¹⁵
ELF	Executable and Linkable Format
EMF	Eclipse Modeling Framework
EPL	Eclipse Public License
EUID	Effective UID
GB	Gigabyte
GCC, gcc	Gnu C Compiler
GMT	Greenwich Mean Time
GNU	GNU's Not Unix
PGP	Gnu Privacy Guard
GPL	General Public License
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HeLP-CCS	Host-Level Protection of Critical Computerized Systems

¹⁵ It seems the “X” was chosen as a filler by opposition with the P (pointer), I (index), S (segment) and F (flag) registers (see e.g. [85]).

HTTP	Hyper-Text Transmission Protocol
HW	Hardware
ID	Identifier
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
IP	Instruction Pointer Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IPC	Inter-Process Communication
ISO	<i>not an acronym</i> (International Organization for Standardization)
JDK	Java Development Kit
JNA	Java Native Access
JNI	Java Native Interface
JTC	Joint Technical Committee
JUL, jul	<code>java.util.logging</code>
JVM	Java Virtual Machine
KPID	Kernel PID (<i>as opposed to the VPID</i>)
KTID	Kernel TID (<i>as opposed to the VTID</i>)
KUID	Kernel UID (<i>as opposed to the VUID</i>)
KVM	Kernel-based Virtual Machine
LGPL	Lesser General Public License
LLC	Last Level Cache
LLVM	<i>originally</i> Low Level Virtual Machine
LTS	Long-Term Support
LT	Linux Trace Toolkit
LT	Linux Trace Toolkit next generation
LT	Linux Trace Toolkit Viewer
LWN	<i>originally</i> Linux Weekly News
MAWG	McAfee Web Gateway
MCCS	Mission Critical Cyber Security
MIT	Massachusetts Institute of Technology

MSB	Most-Significant Byte
NMI	Non-Maskable Interrupt
NIS	Network Information Service
OS	Operating System
PGP	Pretty Good Privacy
PIC	Position-Independent Code
PID	Process Identifier
PMU	Performance Monitoring Unit
POSIX	Portable Operating System Interface
PPA	Personal Package Archive
R & D	Recherche et développement
R&D	Research and Development
RCU	Read-Copy-Update
RDDC	Recherche et développement pour la défense Canada
RFC	Request for Comments
RISC	Reduced Instruction Set Computing
RPM, rpm	<i>originally</i> Red Hat Package Manager
RUID	Real UID
SC	Sub-Committee
SCSI	Small Computer System Interface
SDK	Software Development Kit
SKS	Synchronizing Key Servers
SLF4J	Simple Logging Façade for Java
SSH	Secure Shell
SUID	Saved UID
SW	Software
TCP	Transmission Control Protocol
TID	Thread Identifier
TMF	Tracing and Monitoring Framework
TSDL	Trace Stream Description Language
UDP	User Datagram Protocol
UID	User Identifier

URCU	User-space Read-Copy-Update
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UST	User-Space Tracing
UT	Universal Time
UTC	Coordinated Universal Time
UTS	Unix Time-sharing System
VM	Virtual Machine
VPID	Virtual PID (<i>as opposed to the KPID</i>)
VPID	Virtual Processor ID (<i>Kernel-based Virtual Machine (KVM) context</i>) [84]
VTID	Virtual TID (<i>as opposed to the KTID</i>)
VUID	Virtual UID (<i>as opposed to the KUID</i>)
WG	Working Group

Glossary

Words in *italics* refer to other glossary entries.

Buffering scheme

A template describing how, in a given *domain*, each *channel* should be multiplied according to some pertinent parameter, as well as the number and size of the *sub-buffers* making up each instance. For instance, in the user-space domain buffer multiplication can be done either per-user-ID or per-process-ID. For a given channel, the buffering scheme further specifies the number and size of the sub-buffers making up each buffer. Multiplication per CPU is implicit in all buffering schemes.

Channel

A named arbitrary grouping of *event types*. The channel scope is the *domain*; tracing can be activated and deactivated on a per-channel basis (there are other control modalities as well). Each channel has an associated *buffering scheme*, the leaves of which are sets of buffers, one buffer per CPU. Each of the buffers is made up of *sub-buffers* and maps to a *trace file* (or set of files if chunked).

Consumer, Consumer daemon

A daemon whose single task is to read *tracing session* buffers and write the recovered event records to local or remote storage. Reading a *sub-buffer* frees it for eventual re-use by a *tracer* (except in the *snapshot* case). There can be up to three consumer daemons per *session daemon*: one for the kernel domain, and a pair of 32- and 64-bit consumers for user-space. Each consumer daemon may service several tracing session buffers at once: all of the buffers of the active *channels* of each active session of its owning session daemon for its *domain* and bitness. Recall that each channel may have as many buffers as there are CPUs times the number of active users or processes (depending on the *buffering scheme*).

Domain

A top-level subdivision of the tracing parameter space. Originally, these were hierarchical protection domains, matching the operating system's hardware-enforced privilege levels: kernel and user-space. Now each domain is simply characterised by the existence of specific restrictions on the possible tracing commands and tracing parameter values. For example, *events* from the kernel domain can only be assigned to a *channel* once, whereas events from the user-space domain can each be assigned to multiple channels. The possible *buffering schemes* depend on the domain, etc. Event and channel naming scopes match domains: a channel name in one domain refers to a separate object than the same channel name in another domain, and so on.

Event, Event occurrence

An occasion where an execution thread stepped into a `tracepoint` call. Besides `tracepoints`, events can also be generated by other facilities such as `syscall` (which produces an event every time a system call service routine is entered and exited) and `kprobe/kretprobe` (which produces an event every time its interrupt is serviced).

Event field

Each of the logical units that make up an event record. Each event field has a name and a value. Some event fields are themselves records (C structures) and there may be several levels of such nesting. Non-record fields are primitive data types (integers, floats, characters) or simple arrays of primitives. The event's *payload*, when there is one, is contained by the `event.fields` event field and appears last in each event record.

Event name

A unique identifier within the containing *tracepoint provider*'s scope that serves to specify the *event type*. In the user-space domain, the fully-qualified event name consists of the provider name and the event name; in the kernel domain, the provider is implicit. During tracing, events can be turned on and off individually (there are other control modalities as well).

Event packet

The contents of an individual tracing *sub-buffer*, minus any trailing padding, consisting of contiguous event records. The event records of an event packet are in strictly increasing timestamp sequence. The event packet is the *consumer daemon*'s unit of data transmittal.

Event stream bundle

The set of event streams issuing from all CPUs for a given *event name*. The per-CPU event streams are directed to distinct buffers and eventually wend their way to distinct *trace files*, even though all *event occurrences* are handled by the same *consumer daemon* thread. Event enabling and disabling acts over the event stream bundle. Currently, LTTng provides no means of selectively tracing or filtering CPUs: the datum is available in the resulting trace, however, and CPU filtering can thus be done in post-processing.

Event trace

A time-ordered description of each external input and the time at which it occurred. [3] **This definition is not used in the LTTng context.**

Event type

An abstract description of an *event (occurrence)*'s expected record structure. Each event (occurrence) is an instance of an event type. Each event type is identified by its fully qualified *event name* and is handled by one or more *tracepoint providers*.

Instrumentation

The process of adding intrinsic probes to the source code of an executable.

Payload

The set of *event fields* that is peculiar to that *event type*, if any. An event type may be payload-less. The payload is user-defined (programmed when designing the *tracepoint provider*) and appears as the last field of the event record when present.

Relay daemon

A specialised *consumer daemon* that receives instructions and tracing data from remote systems. Its only task is to commit the tracing data it receives to its system's local storage. It is independent of any local *session daemons*. There may be several relay daemons running on the same system, as long as they use distinct command and data communication ports.

Session daemon

The central session manager process. It sets up, runs and tears down *tracing sessions*, manages the *consumer daemons*, instructs *tracepoint providers* to activate and deactivate their tracepoints, etc. There are at most one root session daemon plus one local session daemon per user. All session daemons operate independently of each other. The local session daemons can only trace their respective local user-spaces, while the root session daemon can trace the kernel and all user-spaces.

Session folder

A folder rooting a hierarchy, the leaves of which are *trace folders*. A session folder matches a particular *tracing session*. This is, by default, reflected in the session folder's name, which is prefixed by the *session name*.

Session name

An identifier given by the user to the *tracing session* (or supplied by default). The *trace name* is constructed from the session name by suffixing it with a one-second resolution timestamp. Session names may be re-used freely over time by any single *session daemon*, and can be used simultaneously by several session daemons as long as they do not share the same \$HOME.

Snapshot

A *trace* reflecting the contents of the tracing buffers at a given instant. When a trace is in snapshot mode, its buffers capture *events* but no committing to storage occurs until a *snapshot record* instruction is issued. Depending on the trace configuration and level of activity, not all of the events in the buffers may make it to storage.

Sub-buffer

A continuous block of memory wherein event records may be written. Sub-buffers have byte sizes which are whole powers of two for addressing and indexing convenience. Since event records are also written in continuous fashion, sub-buffers sometimes need to be padded. Each sub-buffer can be written simultaneously by several writers (which atomically reserve slots in the sub-buffer in order to avoid any overlap) or read by a single reader. Reading and writing are mutually exclusive. Readers (*consumers*) normally read each sub-buffer as a single block, called an *event packet*. Sub-buffers are organised into buffers by a *buffering scheme*.

Trace

(1) [noun] A record of the execution of a computer program, showing the sequence of instructions executed, the names and values of variables, or both. [...] (2) [verb] To produce a record as in (1). [1] [2] [3]

In the LTTng context, a record of the activity of a computer system over a span of time, showing the sequence of instructions executed and the names and values of pertinent variables at the time of each instruction's execution. More narrowly, a time-ordered sequence of event records, manifested as a set of *trace files* and *trace file chunks* organised within a single *trace folder*.

Trace file

A sub-set of the event records defined by a common *channel* and CPU ID. The *metadata* file is a special trace file which holds the trace's metadata, that is to say the event data representation (byte ordering, bitness, etc.), host name, domain, tracer name and version, type of clock, and so on.

Trace file chunk

A *trace file* holding the event records for a partial span of the *trace*'s entire duration. By default, a trace consists of single-chunked trace files, but may be broken up into multiple non-overlapping chunks. Chunks are sequentially numbered but the configuration can be that of a circular buffer, meaning the earliest chunk is not necessarily the first. Trace file chunks are self-consistent: deleting a number of chunks will create gaps in the trace's time coverage but won't make the remaining chunks unusable —chunking is not file-splitting.

Trace folder

A folder containing a set of *trace files* (including in particular a *metadata* file). By default, trace folders are organised under a user's `$HOME/lttng-traces` folder to segregate the trace folders by origin (local or remote), session start time, snapshot time (if applicable), tracing *domain* and sub-domains (e.g. the user-space domain can be broken down by user ID and bitness or by process instance ID). See Section 2.6.2 for the details.

Trace name

A folder identifier constructed from the *session name* by suffixing it with a one-second resolution timestamp. Trace names are supposed to be unique (to the storage device, at least).

Trace root folder

The folder that will hold the *session folders*. By default, this is `$HOME/lttng-traces`. Local traces may be stored in other, arbitrary locations (see the `output` option of the `create` command, Section B.4.3). Traces streaming in from remote hosts are stored in “hostname” folders, themselves stored under the trace root folder. Currently, the *relay daemon* does not allow arbitrary session folder paths.

Tracepoint

A type of intrinsic probe that can be dynamically activated and deactivated by the tracing session manager (*session daemon*). A tracepoint invocation identifies the *tracepoint provider* used and the *event type*, and passes a number of references to contextual entities as arguments. The values of the latter are generally (but not necessarily) stored in the event record, and may also be used to control whether or not the event is recorded —either intrinsically to the tracepoint’s implementation or through a session-manager-provided filter expression. LTTng also encapsulates other tracing mechanisms (`kprobes`, `syscalls`) as tracepoints.

Tracepoint provider

A *tracer* handling a *tracepoint*. Other tracers include the `syscall` and `kprobe` services; these can be managed separately from LTTng but also have encapsulations provided by LTTng. In user-space, tracepoint providers may be statically embedded in instrumented applications or reside in shared objects (dynamic libraries). In the kernel *domain*, the tracepoint providers take the form of kernel modules.

Tracer

(software) A software tool used to trace. [85] [2]

In the LTTng context, the parts of code that generate event records whenever there is an event occurrence. Tracers operate in the context of the traced processes and threads. The *session daemon* merely configures the tracers; the tracers write into the tracing buffers; the *consumers* read from the buffers (generally—but not always—freeing them up in the process) and write to the network or to local storage.

Tracing session

The process of defining, starting, pausing, adjusting and concluding a *trace*. As a data object, the tracing session is a dynamic set of tracing parameters defining the *event types*, their groupings in *channels* within *domains*, the storage instructions and which events and channels are active at any given moment.

This page intentionally left blank.

DOCUMENT CONTROL DATA			
(Security markings for the title, abstract and indexing annotation must be entered when the document is Classified or Designated)			
<p>1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g., Centre sponsoring a contractor's report, or tasking agency, are entered in Section 8.)</p> <p>DRDC – Valcartier Research Centre Defence Research and Development Canada 2459 route de la Bravoure Québec, QC G3J 1X5 Canada</p>		<p>2a. SECURITY MARKING (Overall security marking of the document including special supplemental markings if applicable.)</p> <p>UNCLASSIFIED</p> <p>2b. CONTROLLED GOODS (NON-CONTROLLED GOODS) DMC A REVIEW: GCEC DECEMBER 2012</p>	
<p>3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)</p> <p>LTTng: The Linux Trace Toolkit Next Generation: A Comprehensive User's Guide (version 2.3 edition)</p>			
<p>4. AUTHORS (last name, followed by initials – ranks, titles, etc., not to be used)</p> <p>Thibault, Daniel U.</p>			
<p>5. DATE OF PUBLICATION (Month and year of publication of document.)</p> <p>January 2016</p>		<p>6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.)</p> <p>296</p>	<p>6b. NO. OF REFS (Total cited in document.)</p> <p>85</p>
<p>7. DESCRIPTIVE NOTES (The category of the document, e.g., technical report, technical note or memorandum. If appropriate, enter the type of report, e.g., interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)</p> <p>Scientific Report</p>			
<p>8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)</p> <p>DRDC – Valcartier Research Centre Defence Research and Development Canada 2459 route de la Bravoure Québec, QC G3J 1X5 Canada</p>			
<p>9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)</p> <p>15bl</p>		<p>9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)</p>	
<p>10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)</p> <p>DRDC-RDDC-2016-R9999</p>		<p>10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)</p>	
<p>11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)</p> <p>Unlimited</p>			
<p>12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected.)</p> <p>Unlimited</p>			

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

The Linux Trace Toolkit Next Generation (LTTng) software tracer was thoroughly documented from a user's perspective. The behaviour of each command was tested for a wide range of input parameter values, installation scripts were written and tested, and a variety of demonstration use cases created and tested. A number of bugs were documented as a result (ranging from "simply broken" to "what exactly are we trying to do here?") which are being addressed as time passes and the project keeps advancing in capability, reliability, and ease-of-use.

Whereas the companion *LTTng Quick Start Guide* assumed the simplest installation and use scenario, the *LTTng Comprehensive User's Guide* delves into the nitty-gritty of using LTTng in all sorts of ways, leaving no option undocumented and giving detailed, step-by-step instructions on how to instrument anything (applications, libraries, kernel modules, even the kernel itself) for use with LTTng.

The Linux Trace Toolkit Next Generation (LTTng) offers reliable, low-impact, high-resolution, system-wide, dynamically adaptable, near-real-time tracing of Linux systems. Being able to observe what goes on within the operating system and any applications running on it is an essential prerequisite for the cyber security of governmental and military computer systems of all scales. As the LTTng project has now reached a reasonable level of maturity and can be used in production contexts, it was felt necessary to create a document detailing in practical terms how to deploy, use, and extend it. Large parts of this document have already been used to create the LTTng on-line documentation (<http://lttng.org/docs/#>); this publication can only make LTTng more accessible to all interested parties.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g., Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

LTTng; Tracing; Linux