# Tracing, monitoring and analysis of distributed multi-core systems

*Selected feasibility studies*

In alphabetical order:

DRDC Valcartier
M. Couture

École Polytechnique de Montréal
M. Dagenais
F. Prenoveau

Université Laval
B. Ktari
F. Lajeunesse-Robert

**Defence R&D Canada – Valcartier**

Technical Report
DRDC Valcartier TR 2008-300
April 2009

Principal Author

Mario Couture

Defence scientist

Approved by

Guy Turcotte

SdS Section head

Approved for release by

Christian Carrier

Chief scientist

This work was done between December 2007 and September 2008 under work unit 15BZ10.

# Abstract

Monitoring, tracing and analysis of software execution are indispensable activities for the surveillance, protection and optimization of our national computing infrastructures. However, in recent years, our ability to monitor and analyze software execution has been seriously disrupted by the emergence of multi-core CPUs and the higher level of interconnectivity (between networked systems). These complex systems are already being deployed in our command and control operations. They operate at a much higher transaction rate and they fragment and execute their computing and communication tasks in parallel, leading to huge and extremely complex execution traces to analyze. Current analysis technology is thus overwhelmed by the new computing capability of these systems.

Studies were conducted in 2008 in order to better define the next R&D effort that will be undertaken to address this problem. Among other things, the studies examined the feasibility of developing a feedback-directed diagnostic system based on the *LTTng* framework. Important risks associated with critical technical aspects of this R&D effort were identified and reduced. This document describes the work that was done as well as results and recommendations resulting from these feasibility studies.



# Résumé

Le suivi, le traçage et l'analyse de l'exécution logicielle sont des activités indispensables pour la surveillance, la protection et l'optimisation dans le cadre de notre infrastructure nationale informatique. Néanmoins, ces dernières années, notre capacité à suivre et analyser l'exécution logicielle a été profondément affectée par l'émergence des unités de calcul (CPU) multi cœurs et le haut niveau d'interconnexion (entre les systèmes mis en réseau). Ces systèmes complexes sont déjà déployés dans nos opérations de commandement et contrôle. Leur fonctionnement implique des taux de transaction plus élevés ; ceux-ci fragmentent et exécutent leurs tâches de communication et de calcul en parallèle, ce qui résulte en d'énormes traces d'exécution complexes à analyser. La technologie d'analyse actuelle est donc dépassée par la nouvelle capacité de calcul de ces systèmes.

Des études ont été réalisées en 2008 afin de mieux définir les prochains efforts de R et D qui seront déployés dans le but d'aborder ce problème. Parmi celles-ci, la possibilité de développer un système de diagnostic basé sur la rétroaction et utilisant l'environnement *LTTng* a été étudiée. Les risques importants associés aux aspects critiques de cet effort de R et D ont été identifiés et réduits. Ce document énumère et décrit le travail réalisé ainsi que les résultats et recommandations résultant de ces études de faisabilité.

This page intentionally left blank.

# Executive summary

## Tracing, monitoring and analysis of distributed multi-core systems: Selected feasibility studies

**Couture, M.; F. Lajeunesse-Robert; F. Prenoveau; M. Dagenais; B. Ktari; DRDC Valcartier TR 2008-300; Defence R&D Canada – Valcartier; April 2009.**

Modern military operations rely on sophisticated computing infrastructures that involve: 1) geographically distributed networked topologies supporting collaboration among distributed officers and 2) computational nodes having simple processors, symmetric or asymmetric multi-processors (SMP/ASMP), non-uniform memory access (NUMA) and more recently multi-core (SMP/ASMP on a single chip) systems. This ever-increasing complexity along with disruptions from the complex and often hostile environments within which they interact makes their reliability extremely challenging to guarantee. The current analysis technology for surveillance, protection, optimization and debugging of these systems is overwhelmed by this complexity. Significant improvements and new developments must be put forward in order to improve both in-operation and in-laboratory deep system analysis.

This objective is directly aligned with the following Defence S&T Strategy Technological Challenges (DRDC, 2006);

- *1.5 - Software protection and countermeasures; and*
- *4.6 - Improvements in multi-purpose capability of new and existing systems.*

Studies were conducted in 2008 with the goal of defining the R&D effort that will be required in this domain in order to improve surveillance, protection, optimization and debugging of such complex systems (Dagenais, 2008a; Couture et al., 2008). Six main areas of R&D were identified: *1) Adaptive fault probing, 2) Multi-level, multi-core distributed trace synchronization, 3) Trace abstraction, analysis and correlation, 4) Automated fault identification, 5) System health monitoring and corrective measure activation, and 6) Trace directed modelling.* A joint project proposal involving Ericsson Canada, NSERC, DRDC Valcartier as well as a number of experts from different Canadian universities was then proposed to NSERC and DRDC for funding in 2008 (Dagenais, 2008b; Couture and Charpentier, 2008). The tracing technology that was considered in this proposal was the *LTTng* framework (LTTng, 2008).

The 2008 studies included a detailed analysis of critical technical aspects of the proposed project in order to identify and reduce major risks prior to the beginning of the proposed R&D project. They focused on the feasibility of developing a feedback-directed diagnostic system based on the *LTTng* framework. This document describes the work that was done and the results and recommendations of these feasibility studies.

The work described in this document is divided into two main parts (R&D threads). They are:

- **R&D Thread 1: Mechanisms for monitoring, tracing and control**. The aim of this thread was to verify the feasibility of extending the *LTTng* framework for: 1) network transfer of

execution traces, 2) latency transfer minimization, 3) *LTTng* remote control, and 4) continuous decoding of execution traces.

- **R&D Thread 2: Toward trace abstraction and analysis**. The aim of this thread was to: 1) review the current work and technology in this domain, 2) analyze execution traces, 3) study the feasibility of abstracting and analyzing execution traces, and 4) study the feasibility of developing automatic mechanisms for trace analysis.

Results and observations from R&D Threads 1 and 2 clearly show the great potential of the proposed R&D project. Each of the studied components (for the online monitoring, tracing and analysis of distributed multi-core systems) is clearly feasible and very promising. Four important challenges must be addressed. Actually, they can be considered as recommendations for the next R&D efforts.

1. First, the overhead associated with the *LTTng* framework should be as low as possible to ensure it is widely applicable and effective.

2. The second important challenge is to facilitate the integration of the different sources of data (e.g. static and dynamic, kernel and user level trace points) from different processor cores, virtual and physical machines and distributed systems. Moreover, the database structure must allow the addition of synthesized information to the event traces (e.g. state information, abstract higher level events, etc.).

3. The third important challenge is related to the analysis of execution traces in real operational situations. Traces obtained from concurrent and parallel processes which are executed on distributed multi-core CPUs and make accesses to shared resources, must be analyzed. The "resolution issue" must be addressed as well.

4. Finally, another important challenge consists in finding the most appropriate formalism for trace analysis, trace abstraction and fault detection. The Kleene algebra formalism is adequate for the certification process but not for the efficient online handling of huge execution traces. A formalism to solve these specific problems must be identified or developed.

# Sommaire

## Tracing, monitoring and analysis of distributed multi-core systems: Selected feasibility studies

**Couture, M.; F. Lajeunesse-Robert; F. Prenoveau; M. Dagenais; B. Ktari; DRDC Valcartier TR 2008-300; R & D pour la défense Canada – Valcartier; Avril 2009.**

De nos jours, les opérations militaires dépendent d'infrastructures informatiques qui impliquent : 1) des topologies de réseau géographiquement réparties qui facilitent la collaboration entre officiers et 2) des nœuds de calcul comprenant des processeurs de calcul simples, symétriques, asymétriques (SMP/ASMP), des accès non uniformes à la mémoire (NUMA) et, plus récemment, les systèmes à processeurs multicœurs (systèmes SMP/ASMP sur une puce). Cette complexité toujours grandissante, de concert avec l'influence négative des environnements hostiles dans lesquels ces systèmes évoluent, rend leur fiabilité très difficile à garantir. La technologie d'analyse réalisant la surveillance, la protection, l'optimisation et le débogage de ces systèmes est maintenant rendue dépassée par cette complexité. Des améliorations significatives et de nouveaux développements doivent être mis de l'avant dans le but d'améliorer l'analyse système qui est réalisée tant sur le terrain qu'en laboratoire.

Cet objectif est directement aligné sur les défis technologiques de la défense (DRDC, 2006) :

- *1.5- Software protection and countermeasures; and*
- *4.6- Improvements in multi-purpose capability of new and existing systems.*

Des études ont été réalisées en 2008 dans le but de définir l'effort de R et D qui sera fait dans ce domaine afin d'améliorer la surveillance, la protection, l'optimisation et le débogage de ces systèmes complexes (Dagenais, 2008a ; Couture et al., 2008). Six directions de R et D ont été identifiées; elles sont : *1) Adaptive fault probing, 2) Multi-level, multi-core distributed traces synchronization, 3) Trace abstraction, analysis and correlation, 4) Automated fault identification, 5) System health monitoring and corrective measure activation and 6) Trace directed modeling*. Un projet conjoint impliquant Ericsson Canada, CRSNG, RDDC Valcartier et plusieurs experts provenant de différentes universités canadiennes a été proposé au CRSNG et à RDDC pour financement en 2008 (Dagenais, 2008b; Couture et Charpentier, 2008). La technologie de traçage qui a été avancée dans cette proposition est celle de l'environnement *LTTng* (LTTng, 2008).

Parmi les études de 2008, l'analyse détaillée des aspects critiques du projet proposé a été réalisée dans le but d'identifier et de réduire les risques majeurs, et ce, avant le début du projet de R et D proposé. Leur centre d'intérêt a été orienté sur la possibilité de développer un système de diagnostic basé sur la rétroaction et l'utilisation de l'environnement *LTTng*. Ce document énumère et décrit le travail réalisé ainsi que les résultats et recommandations résultant de ces études de faisabilité.

Le travail décrit dans ce document est divisé en deux parties principales (Sujets R et D) qui sont :

- **Sujet R et D 1 : Mécanismes pour le suivi, le traçage et le contrôle**. Ce sujet avait pour but de vérifier la possibilité d'étendre l'environnement *LTTng* pour : 1) le transfert réseau

des traces d'exécution, 2) la minimisation de la latence de transfert, 3) le contrôle distant de *LTTng* et 4) le décodage continu des traces d'exécution.

- **Sujet R et D 2 : Vers l'abstraction et l'analyse des traces**. Ce sujet avait pour but de : 1) faire une revue des travaux et de la technologie courante dans ce domaine, 2) analyser des traces d'exécution, 3) étudier la possibilité d'abstraire et d'analyser des traces d'exécution et 4) étudier la possibilité de développer un mécanisme automatique d'analyse de traces.

Les résultats et observations découlant des Sujets R et D 1 et 2 montrent clairement tout le potentiel du projet proposé. Chacune des composantes étudiées (pour le suivi, le traçage et l'analyse en ligne des systèmes répartis multicoeurs) est clairement réalisable et très prometteuse. Quatre défis importants doivent être surmontés. En fait, ils peuvent être considérés comme étant des recommandations pour le prochain effort de R et D.

1. Premièrement, la charge de travail CPU associée à l'environnement de *LTTng* devrait être réduite autant que possible afin de lui assurer une applicabilité et une capacité étendue.

2. Le second défi important consiste à faciliter l'intégration des différentes sources de données (statiques, dynamiques, des points de traçage dans les « kernel/user spaces ») provenant de différents cœurs de processeurs, de machines virtuelles et physiques, et de système répartis. De plus, la structure de la base de données utilisée doit permettre l'ajout d'information de synthèse aux éléments des traces d'exécution (états, abstractions haut niveau, etc.).

3. Le troisième défi important est lié à l'analyse des traces d'exécution dans des situations opérationnelles réelles. Les traces obtenues de processus concurrents et parallèles, qui sont exécutés sur des systèmes multicœurs et qui font des accès aux ressources des systèmes, doivent être analysées. Le problème de la « résolution » doit être également abordé.

4. Finalement, un autre défi important consiste à trouver le formalisme le plus approprié pour l'analyse des traces, l'abstraction des traces et la détection de fautes. Le formalisme de l'algèbre de Kleene est adéquat dans les processus de certification, mais il ne l'est pas pour la manipulation en ligne efficace des traces d'exécution qui sont énormes. Un formalisme doit être identifié ou développé pour aborder ces problèmes particuliers.

# Table of contents

# List of figures

# List of tables

# Acknowledgements

The authors would like to thank DRDC Valcartier/SoS Section and MITACS for the financial support which allowed two graduate students to be hired for a four-month internship.

This page intentionally left blank.

# 1 Improving robustness of command and control information systems

Information systems became much more complex in the last decade in terms of technology evolution and the more demanding needs related to command and control (C2). Actually, their architecture is mainly determined by organizational forms of C2. Some important characteristics of current military C2 are given by Benaskeur and Blosgett (2008):

- *C2 is a distributed environment;*
- *C2 has functional architectures such as surveillance, threat evaluation, etc.;*
- *C2 is a complex process;*
- *C2 deals with large volumes of data under stringent time constraints;*
- *C2 can be influenced by deterministic/stochastic, episodic/sequential, static/dynamic environments; and*
- *the decision-making process may be based upon single/multiple criteria.*

C2 can also be considered as: *reactive (self-maintaining: fixed finality)*, *responsive (goal seeking: variable but determined finality)* or *active* (or pro-active) *(purposeful: variable and chosen finality)* (Couture, 2007). The complexity of supporting information systems arises from (among other factors):

- *the interaction of the non-functional requirements such as maintainability, performance, and security* (Ellison and Woody, 2007b);
- the higher level of interconnectivity between physically distributed C2 information systems;
- the use of computational nodes involving simple processors, symmetric or asymmetric multi-processors (SMP/ASMP), non-uniform memory access (NUMA) and more recently multi-core systems (SMP/ASMP on a single chip)[1].

These characteristics are not unique to military domains. Actually, C2 often takes place (under different forms) in civilian organizations as well. For instance, the operational management of the complex computational infrastructure used by a telephony company is another similar complex C2.

This complexity along with disruptions from complex and often hostile environments in which they are interacting makes their reliability extremely challenging to guarantee. The current analysis technology used to ensure effective and efficient surveillance, protection, optimization and debugging of these systems during operation is overwhelmed by this complexity. Significant improvements and new developments must be made in the concepts, algorithms and techniques in order to achieve and maintain the needed survivability capabilities in operation (Avizienis et al. 2004; Neumann 2000; Neumann 2004).

[1] A recent survey (Heikkila and Gulliksen, 2007) mentions that communications and military/aerospace are the most probable early adopters of multi-core processor boards.

## 1.1 Context of this work

According to the methodology used[2], preliminary studies and workshops were conducted in 2008 in order to define the R&D effort that will be required in the domain of monitoring, tracing and analysis of distributed multi-core systems (Couture et al., 2008; Dagenais, 2008a). Six main areas of R&D (herein called **Areas of R&D**) were identified[3]: *1) Adaptive fault probing, 2) Multi-level, multi-core distributed trace synchronization, 3) Trace abstraction, analysis and correlation, 4) Automated fault identification, 5) System health monitoring and corrective measure activation, and 6) Trace directed modelling.*

A joint project involving Ericsson-Canada, NSERC, DRDC Valcartier as well as a number of experts from different Canadian universities was then proposed to NSERC and DRDC for funding in 2008 (Dagenais, 2008b; Couture and Charpentier, 2008). The main technology that was considered in this proposal was the *LTTng* framework (Desnoyers and Dagenais, 2006a; Desnoyers and Dagenais, 2006b; LTTng, 2008).

Investigations of critical aspects of the proposed project were also conducted in 2008 in order to identify and reduce major technical risks. Their purpose was to study the feasibility of developing a feedback-directed diagnostic system based on the *LTTng* framework. This document describes the results and recommendations of these studies.

## 1.2 Overarching methodology

The methodology for defining DRDC Valcartier R&D projects consists of a suite of logically ordered phases (Figure 1), each of which builds on the results of the previous phases. This section provides a brief overview of each phase and establishes relationships with the current R&D effort.

The first two phases are dedicated to the formulation of operational and technical needs and requirements. The process starts with the identification and description of operational problems. Needs and requirements are then iteratively refined by scientists with the active contribution of representatives from the end-user community and technology specialists (two opposing arrows between phases 0 and 1 in Figure 1).

Based upon the results of the first two phases, scientists identify a number of potential solutions out of the scientific literature (phase 2). They typically require in-depth scientific investigations to be critically reviewed and validated.

The third phase of the process consists in achieving a state-of-the-art[4] (SOTA) regarding the problems, needs and solutions identified. The SOTA should allow the capture and integration of all relevant concepts, research, authors, organizations, technologies, processes, tools, etc., into a comprehensive view.

[2] Described in Section 1.2.
[3] Areas of R&D are briefly described in Annex B.
[4] A SOTA on the monitoring and tracing of multi-core systems was made (Dagenais, 2008a).

In phase 4, international experts, civilian and military partners of DRDC, and academic and industry stakeholders are invited to elaborate on selected aspects of the identified solutions[5]. The main goals of the tutorial are:

1. distribute the acquired knowledge among stakeholders;

2. develop a more precise understanding of identified problems and needs;

3. provide complete answers to attendees' questions; and

4. identify, define and start collaborations among stakeholders.

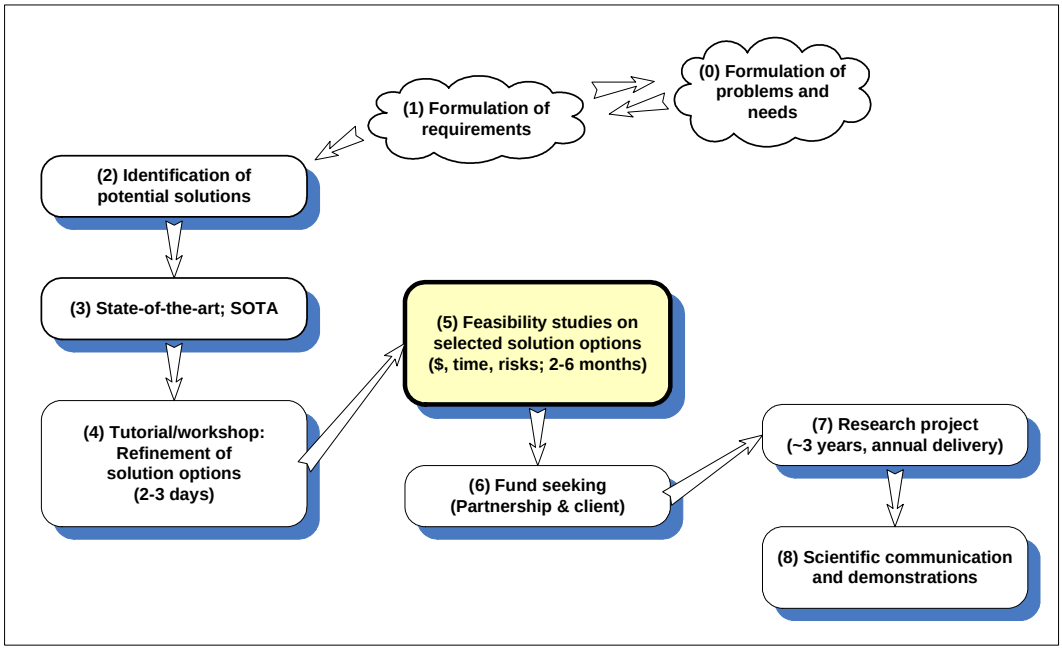


*Figure 1. The Methodology phases.*

Just after the tutorial, a half-day workshop involving a limited number of partners is then held. Based upon acquired knowledge, participants discuss more specific problems and needs in greater depth. The aim of this workshop is to prioritize a number of solution options for subsequent feasibility studies.

In phase 5, a number of feasibility studies are carried on in order to estimate the cost and technological risks that are associated with each selected solution option[6]. Based upon these results, R&D projects are then formally proposed for funding in phase 6[7], and then executed (phase 7). Ultimately, they lead to demonstrations in realistic environments, and other forms of publication (phase 8).

[5] Couture et al., (2008) provide a description of the tutorial/workshop that was held at Ericsson's Montreal offices in January 2008.
[6] This document represents the main output of this phase for this effort.
[7] Two project proposals were submitted: Couture and Charpentier (2008) and Dagenais (2008b).

The methodology (or process) shown in Figure 1 may be cycled through many times if many iterations are needed to build the complete solution. In this case, each iteration builds on the results of previous ones.

## 1.3 Content of this document

The work described in this document is divided into two separate but complementary threads (herein called **R&D Threads**). When considered together, both threads aim to study the feasibility of building an online feedback-directed diagnostic system for the tracing and analysis of C2 information systems during operation. They are briefly introduced in the following sections.

### 1.3.1 R&D Thread 1: Mechanisms for monitoring, tracing and control

This R&D thread aims to identify and study concepts and mechanisms for the online generation, transfer and remote management of execution traces. Additional mechanisms for the remote control of the tracing software (LTTng, 2008) will also be studied[8]. The aim of this thread is the control-directed remote tracing of distributed software systems. The main tasks defining this thread are summarized below.

1. Identify and validate mechanisms for the transfer of execution traces from one or many distributed multi-core computational nodes to a remote one.

2. Identify and validate mechanisms for the reception and management of execution traces on a remote computational node. Mechanisms should assemble partial execution traces that originate from distributed, SMP and multi-core computational nodes. They should also make them available for analysis processes.

3. Identify and validate mechanisms for the transfer of control commands between computational nodes (for the remote control of *LTTng*).

4. Identify and validate mechanisms for the activation of the control of *LTTng* probes (based on the execution of transferred control commands).

5. Evaluate the performance of all improved and developed mechanisms.

### 1.3.2 R&D Thread 2: Toward trace abstraction and analysis

Tasks defining R&D Thread 2 are complementary to those of the previous thread. Essentially, this thread aims to identify and study mechanisms that will permit effective and efficient remote online analysis of the received data (execution trace elements)[9].

The main tasks defining this thread are summarized below.

[8] Outcomes of these tasks will make a direct contribution to the following Areas of R&D: 1, 2 and 3 (Annex B).
[9] Outcomes of these tasks will make a direct contribution to the following Areas of R&D: 3, 4, 5 and 6.

1. Review the scientific literature and works on formalisms for representing programs, execution traces, fault representation, as well as trace comparison, pattern and fault detection.

2. Identify and evaluate mechanisms that will be used to generate theoretical representations of software applications in the trace analysis context, resulting in execution models.

3. Simplify the trace abstraction and analysis problem by limiting the tracing to the user/kernel interfaces (e.g. system calls). Analyze limitations that are associated with this simplification.

4. Study the content, structure and inherent logic of execution traces.

5. Investigate the possibility of abstracting execution traces into higher-level behaviours. Compare execution traces with execution models in order to correlate observed events with process behaviours.

6. Investigate the possibility of automating trace abstraction and analysis.

## 1.4 Frequently used terms

Frequently used technical terms in this document are defined below. The Glossary contains definitions of other terms.

**Computational node**: a piece of hardware that is running software applications in operation. It may refer to computers, network devices, etc.

**Software application (or program) versus software system**: these terms refer to two similar things that stand at two different conceptual levels. Software application (or program) refers to a piece of software (lower level), while software system refers to an aggregation of one or more software applications (higher level, the whole). Two examples will help clarify the difference between these terms. 1) **Microsoft context**: the software application would be MS Word and the software system would be MS Word plus the MS Windows XP operating system. 2) **Linux distribution context**: the software application would be a specific package and the software system would be the distribution as a whole (including the Linux kernel).

**Software component**: components are smaller pieces of software applications, software systems and operating systems (e.g. a kernel module).

**Information system**: a software system that is used by officers or operators during operations. For instance, it may involve many distributed collaborative software applications that are executed on many geographically distributed computational nodes. In the military context, they may be a command and control information system (C2IS).

**System**: this general term includes any form of any sub-system. A system usually consists of one computational node, which runs software systems.

**Diagnostic system**: a software application that monitors and/or traces executions of software systems. For instance, it may produce diagnostics regarding system health.

**Online and continuous:** the adjective “online” means in this document being in use during operations, and the adjective “continuous” means constant and uninterrupted.

**Execution trace**: execution trace is defined in this document as a chronological suite of records resulting from observations that were made on a running system at different instants $t_n$, during a specific period delta-t. More precisely, an execution trace contains information regarding specific events in the system that is generated by *LTTng* when its active probes were encountered and executed.

**Trace events (or elements)**: the content of an execution trace.

The reader will find additional definitions in the Glossary of this document.

# 2 R&D Thread 1 – Mechanisms for monitoring, tracing and control

This R&D thread aims to identify and study mechanisms for the online generation, transfer and remote management of execution traces. Additional mechanisms allowing the remote control of the tracing software (*LTTng*) will also be studied. The aim of this thread is the **control-directed remote tracing of distributed software systems**. This chapter provides a description of the work, observations and results achieved in this R&D thread.

## 2.1 Initial premises

Main premises considered in this thread are:

1) Once appropriately adapted, the *LTTng* framework[10] will transfer execution traces on a continuous basis to a remote computational node for further management and analysis. Minimal impacts will be observed on traced systems' performance.

2) It will be possible to decode and reassemble, on a continuous basis, chronological elements of execution traces which originate from one or many distributed and possibly multi-core systems.

3) Remote control of each running instance of the *LTTng* framework (located on each distributed system) will be made possible on a continuous basis. The control consists in allowing the choice of both the focus and resolution[11] of execution traces.

## 2.2 Technological considerations

### 2.2.1 The Use of Free Open Source Software – The *LTTng* framework and Linux

The choice of using the *LTTng* framework and the Linux operating system (and more generally FOSS) was made in the early stages of this R&D effort. More information regarding this choice and potential advantages of using FOSS can be found in Couture et al. (2008), Carbone (2006a), Carbone (2006b), Carbone and Charpentier (2006), Carbone (2008) and Charpentier and Carbone (2004).

[10] LTTng was chosen as the tracing system (Couture et al., 2008). A quick overview of this software can be found in Annex C. Some important references regarding LTTng are: Desnoyers and Dagenais (2006a); Desnoyers and Dagenais (2006b); Desnoyers and Dagenais (2008); Yaghmour, K. and M. R. Dagenais (2000); Yaghmour (2001); Bligh et al. (2007).

[11] The **focus** refers to the specific traced software objects or components and **resolution** refers to the number of active probes used. A higher number of well selected active probes will produce well focused execution traces, with the necessary levels of detail for the analysis of its elements.

One advantage (among others) of using the *LTTng* framework is that some of its components may be used with other types of operating system (that are not UNIX-like).

### 2.2.2 Use of the client-server model

The client-server model (Figure 2) was used in this project. This model was chosen for the following reasons.

- **Information systems (clients) may be geographically distributed**: operations may involve the use of spatially distributed networked collaborative information systems. This model is particularly useful for geographically distributed officers that collaborate through networked information systems to achieve shared goals or missions.
- **Information systems (clients) and the software for trace management and analysis (on the server) is not necessarily executed on the same computational node**: critical operations may require that system monitoring, tracing and analysis be achievable both locally and remotely. In some circumstances, remote analysis may be preferred. For instance, intense local trace analysis may seriously impact the performance of the computational node CPU(s). This problem would be solved if the workload associated with trace analysis were achieved remotely, freeing up the CPU(s) for operational computing tasks.

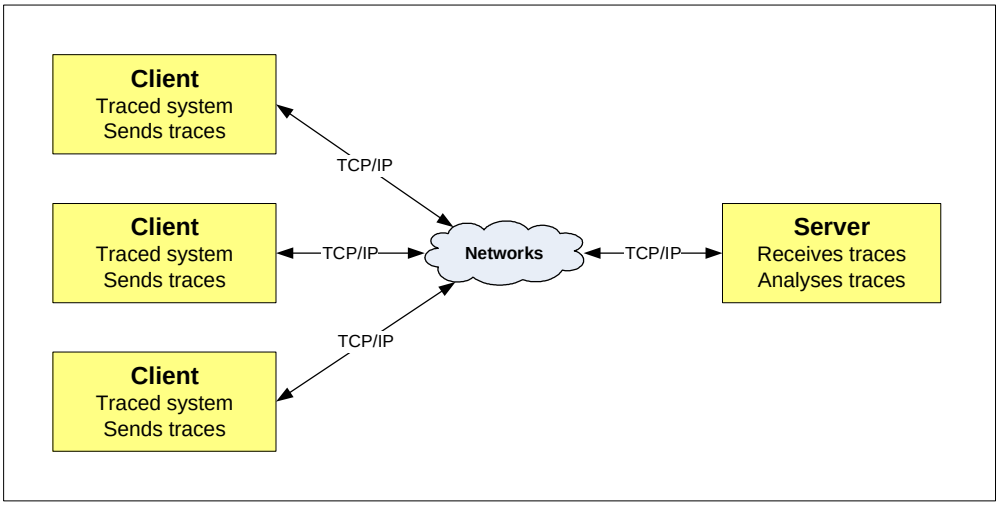


Figure 2. The client-server model.

Using this model, concurrent execution traces (on the clients' side) are transferred to a centralized server. This server reassembles received execution traces and performs appropriate analysis on these traces (or delegates this analysis to other supporting computational nodes). The analysis of these traces characterizes the current health of each traced system as well as the whole distributed system.

### 2.2.3 Prototype architecture

Figure 3 shows a view of the architecture that is studied in this R&D thread (as well as in the next chapter). It involves the use of the client-server model (Section 2.2.2). It allows the study of remote management and analysis of execution traces, as well as remote control of *LTTng* probes on the client's side. The figure only shows one client for clarity purposes, but the reader should keep in mind that many distributed concurrent multi-core clients are considered in this study.

The client (computational node 1) executes user's programs and *LTTng*. In this work, additional components are added on the client side in order to: 1) transfer execution traces to remote server, and 2) receive and execute *LTTng* control commands. The reception of execution traces on the server side (computational node 2) is managed by the receiver component. Traces are then analyzed by other components and the knowledge regarding the health of the client system is produced. Based on this knowledge, a decision (involving humans) is made regarding what should be traced[12] next. Control commands are then generated accordingly; and are sent and executed on the client by the controller component.

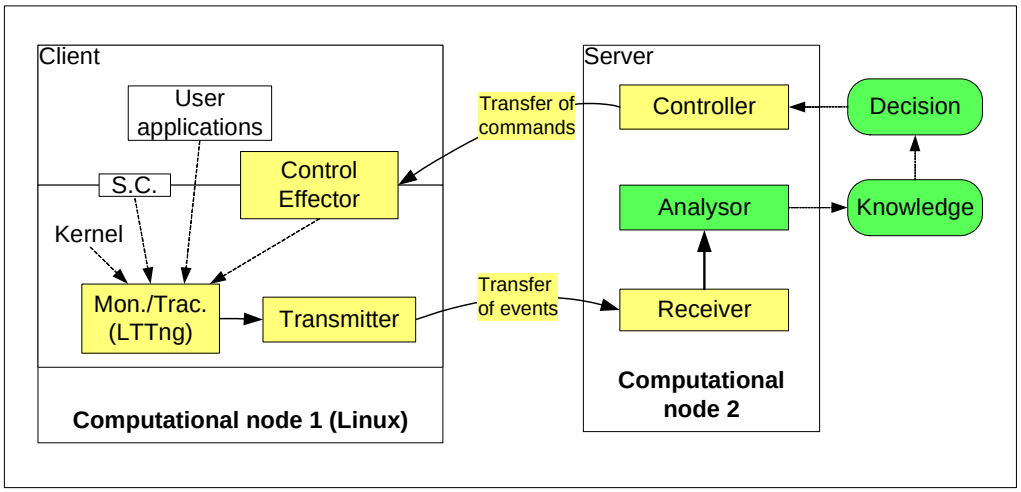


*Figure 3. The prototype architecture used.*

Yellow/green rectangles in Figure 3 are studied in R&D Thread 1/2.

### 2.2.4 Programming language

C programming language was exclusively chosen in this R&D thread for performance and compatibility reasons[13].

[12] Only the control of the LTTng tracing probes is considered in this study. Further studies will consider the possibility of controlling user's programs and the operating system as well. For instance, services could be started or stopped depending on the results obtained.

[13] Both the Linux kernel and LTTng are mainly programmed in C.

### 2.2.5 Computational nodes

Both single-core (Intel Pentium 4, 1.70GHz) and multi-core (Intel Xeon, E5405, 2.00 GHz) computational nodes were used in this R&D thread. They were linked by a 10/100BASE-TX Ethernet switch.

## 2.3 Workload for this R&D thread

Preliminary investigations of the *LTTng* architecture yielded the identification of four main groups of feasibility studies that could be made. They are briefly introduced below.

1. **Support for network**. Currently, the transfer of execution traces for online remote analysis is not allowed. Analysis is limited to complete traces that were already saved in local files on a hard disk (on the client's side). The feasibility of transferring trace elements from clients to a remote server over the network for online remote analysis is studied.

2. **The flushing of data buffers**. Before trace elements can be sent to the server, they are locally sent from the kernel space to user space (client side) via data buffers (using relay; Zanussi et al., 2003; RelayFS, 2008). The fact that these buffers are flushed only when they are full presents a problem. The feasibility of flushing these buffers on a more regular basis in order to reduce the latency of transfer is studied.

3. **Remote control**. The feasibility of remote activation and deactivation of *LTTng* probes (client side) for the remote control of *LTTng* is studied.

4. **The continuous decoding of execution traces**. Currently, execution traces are decoded by *LTTV*. *LTTV* is limited to reading complete execution traces that were pre-recorded on a hard disk. The feasibility of decoding execution traces that are transferred from clients to a server on a continuous basis is studied.

Five main tasks for this R&D thread are identified. They are described below.

**Task 1 – The refactoring of selected *LTTng* components:**

The main goal of this preliminary task consists in facilitating code reuse for subsequent R&D work. *LTTng*'s components are studied in more depth in order to identify the source code that could be refactored, and achieve this refactoring.

**Task 2 – The transfer of execution traces over the network:**

A number of mechanisms must be put in place in order to allow the secure, efficient and effective transmission of execution traces from clients to server for remote analysis and control. This task involves, among other things, the study of transfer protocols and the modification of some *LTTng* components.

**Task 3 – The problem of latency transfer:**

Internally, *LTTng* transfers execution trace events (or trace elements) from the kernel space to the user space through data buffers (client side). These buffers are flushed only when they are full. This mechanism may cause a latency problem if the rate of production of trace elements is not high enough; the data may be kept in buffers for long periods of time. This task aims to identify and test mechanisms that will flush these buffers, and thus reduce the time the data will reside in buffers.

**Task 4 – Remote control:**

Secure, efficient and effective remote control of *LTTng* probes represents an important advantage for the surveillance and protection of information systems in operation. The types of control that will be considered in this R&D thread are limited to the activation/deactivation of selected *LTTng* probes. Mechanisms to ensure such a control are studied and tested. They incorporate security and authentication functionality.

**Task 5 – The decoding of received execution traces:**

Raw execution traces must be decoded and transformed by *LTTV* before they can be analyzed. This task aims to study and test mechanisms for the remote management of execution traces.

## 2.4 Exploration and analysis – Results and observations

### 2.4.1 Technical description of an execution trace

It is important at this point to provide a description of execution traces that are produced by *LTTng*. Terms and concepts that are defined in this section will be used throughout this chapter. The reader may refer to Figure 4 for an illustration of definitions and concepts.

An **execution trace** is defined in this document as *a chronological suite of records resulting from observations that were made on a running system at different instants $t_n$, during a specific period delta-t.* More precisely, an execution trace contains information regarding specific events in the system that is generated by *LTTng* when its active probes were encountered and executed.

Execution traces are sub-divided into **channels**[14]. A channel gathers together events that are generated by a group of pre-selected probes. A **tracefile** is a sequence of events, which takes the form of a tracefile in the execution trace. There is one tracefile per running CPU for each channel. Tracefiles are then sub-divided into a number of fixed size blocks called **sub-buffers**. Each sub-buffer contains many events (or elements).

The concept of channel allows the segregation of traced events according to their importance or priority. If a problem occurs while tracing the execution of a system, the capture of events corresponding to prioritized channels would be captured before other channels.

[14] Currently, LTTng has five channels.

Tracefiles are read by *lttd* process (Section 2.4.2) using a memory map of the sub-buffers. This process makes a reservation for a filled sub-buffer, reads its content and then releases it so that it can be refilled again.

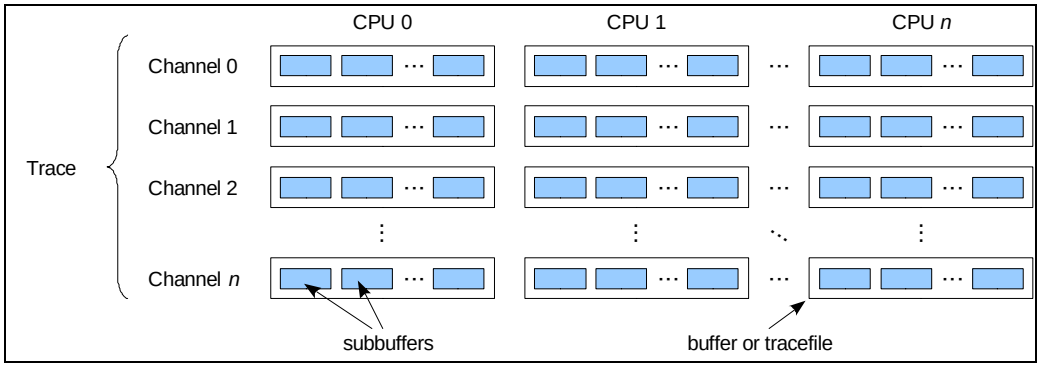


*Figure 4. Composition of a LTTng execution trace.*

Note that *LTTng* is lockless, meaning the *LTTng* tracer may write in sub-buffers even if they are reserved by the process *lttd*. A sub-buffer may thus become "corrupted" if the tracer writes into it while the *lttd* process is reading it. The *lttd* process is notified of a corruption by a status code that is returned when a sub-buffer is released. This status code should be considered by *lttd*, and managed accordingly.

For each channel, *LTTng* offers the possibility of specifying both the size and the number of sub-buffers that are allocated in the kernel.

### 2.4.2 Initial architecture of the *LTTng* framework

Figure 5 provides an overview of some selected components of the current *LTTng* architecture. Modifications and new developments were added to this architecture in order to achieve described feasibility studies. The reader will find a more detailed description of *LTTng* in Annex C. These components are introduced below.

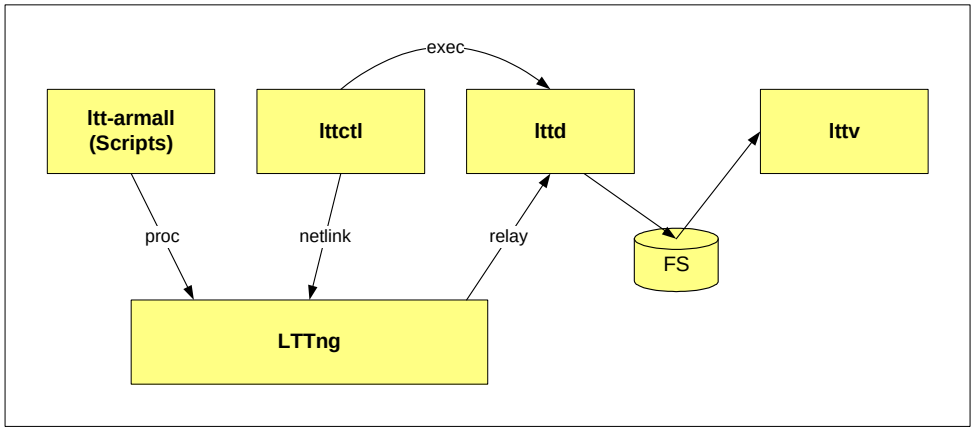


*Figure 5. Main components of LTTng – current version.*

Components *ltt-armall* and *ltt-disarmall* (not shown) are sets of scripts that are used to activate or deactivate probes used by *LTTng*. The mapping of communication channels with probes[15] is achieved by these scripts as well.

The component *lttctl* is a command line application that creates, starts and stops the generation of execution traces through a netlink interface. It also starts the *lttd* daemon, which transfers execution traces (on hard disk) that were received from the kernel space through "relay"[16] (RelayFS, 2008; Zanussi et al., 2003).

Execution traces that are saved on disk can be viewed with the *LTTV* component.

### 2.4.3 Task 1 – Refactoring of selected *LTTng* components

Preliminary investigations of components shown in Figure 5 have shown that there would be benefits associated with the refactoring of *lttd*.

The work that was done in this task has allowed the extraction of the *lttd* "execution trace reading functionality" and to refactor it as a reusable library. The main goal was to make execution traces directly available both locally and remotely for online trace analysis.

The function prototypes that are available in this new library are listed in Figure 6.

```
struct lttio_handle *lttio_open(const char *channel_name, struct lttio_ops *ops, int
flags);

int lttio_start(struct lttio_handle *h, int num_threads);

int lttio_wait(struct lttio_handle *h);

int lttio_hang_up(struct lttio_handle *h);

int lttio_close(struct lttio_handle *h);
```

*Figure 6. The new library API for the acquisition of execution traces.*

Following is a brief description of these functions.

- **`lttio_open()`**: this function opens an execution trace that is located in the directory, which is specified by the `channel_name` argument. It returns a handle that identifies the `liblttio` session and allows the management of acquisitions. The `lttio_ops` data structure is described later in this section.
- **`lttio_start()`**: this function starts the worker threads that are used for the capture of execution traces and returns immediately. The number of started threads is determined by the parameter `num_threads`.

[15] The list of active probes that were used in LTTng throughout this work can be found in Annex C.4.
[16] The old appellation of "relay" was "RelayFS".

- **`lttio_wait()`**: the capture of a trace ends when it has reached its end or when the capture is forced to stop (with the function `lttio_hang_up()`). This function suspends the execution of the calling thread until all the worker threads have ended their execution.
- **`lttio_hang_up()`**: this function sends a hang up request to the worker threads. It does not wait for the threads to stop, it returns immediately.
- **`lttio_close()`**: this function closes all tracefiles and frees all resources that were allocated by the acquisition session.

As shown in the API, the data structure `lttio_ops` must be transmitted as a parameter in the function **`lttio_open()`** (Figure 7). This structure contains pointers to a number of callback functions that are called by the worker threads throughout the trace acquisition process.

```
struct lttio_ops {

    int  (*open) (struct lttio_tracefile *tf);

    int  (*write) (struct lttio_tracefile *tf, void *data, unsigned int size);

    int  (*subbuf_begin) (struct lttio_tracefile *tf);

    int  (*subbuf_end) (struct lttio_tracefile *tf, enum lttio_subbuf_status status);

    void (*close) (struct lttio_tracefile *tf);

};
```

*Figure 7. Data structure used in the library for the capture of execution traces.*

The function `open()` is called when a new tracefile is opened. Then, for each acquired sub-buffer that contains trace events (elements), the function `subbuf_begin()` is called, followed by many calls to the function `write()`. Finally, the function `subbuf_end()` is called[17].

The acquired data in the sub-buffers are transferred to the user by calling the function `write()`. It is important to note that the data may be transferred in the form of many data segments.

The function `subbuf_end()` marks the end of a sub-buffer. This function returns an `int` representing the status of the sub-buffer. The value `LIBLTTIO_SUBBUF_OK` indicates that the sub-buffer was not corrupted during the trace acquisition, and the value `LIBLTTIO_SUBBUF_CORRUPTED` indicates that is was corrupted.

The `lttio_tracefile` data structure that is passed as a parameter in these callback functions is listed in Figure 8. The variable `path` contains the relative path of the channel (e.g. "`control/facilities_0`"), `subbuf_size` represents the size of the current sub-buffer, `subbuf_bytes` indicates the size of the data contained in the current sub-buffer in bytes, and

[17] Note that new tracefiles may appear in the middle of a tracing session when a CPU is hot-plugged on an SMP system.

`user_data` is a pointer that can be used by the user to attach its own data to the tracefile structure.

```
struct lttio_tracefile {

    char *path;

    unsigned int subbuf_size;

    unsigned int subbuf_bytes;

    void *user_data;

};
```

*Figure 8. Data structure used in the library for the acquisition of execution traces.*

### 2.4.4 Task 2 – Transfer of execution traces over the network

As mentioned, the client-server model was chosen. Clients are expected to send execution traces to one server that manages and possibly analyzes them. This model offers the advantage of allowing the remote management and analysis of many simultaneous traces originating from distributed SMP or multi-core systems. The server reassembles received traces and achieves appropriate analysis (or delegates this analysis to other supporting computational nodes).

Three different communication solutions were initially considered for the transfer of execution traces over the network. They are listed below.

1. communication through serial connections;

2. communication through common Ethernet and network devices using UDP/IP; and

3. communication through common Ethernet and network devices using TCP/IP.

The third solution was chosen for many reasons. Some of the advantages it offers are: 1) it offers mechanisms for flow control and the management of transmission errors; 2) it offers a certain level of reliability; 3) it may potentially allow the transmission of execution traces over networks having large geographical extents; and 4) depending on the hardware and network devices used, it offers sufficient bandwidth for the transfer of high resolution (huge) execution traces.

Following is a description of the solution that was put forward for communication between computational nodes.

**Communication between computational nodes:**

A TCP connection is established for each tracefile. Events pertaining to tracefile channels can then be concurrently transmitted.

For each tracefile, the client transfers one header at the beginning of the connexion (Table 1)[18]. Then, sub-buffers of the tracefile are transmitted one after the other until the end of the execution trace is reached. The closing of the TCP connection indicates the end of the execution trace.

The size of each transmitted sub-buffer can be prescribed by the parameter `subbuf_bytes`. This parameter represents an advantage when sub-buffers are not completely filled with the data. Indeed, the value of the parameter `subbuf_bytes` can be used to transmit only sub-buffers' used bytes; unused bytes are not transmitted. It will be shown later in this chapter (Section 2.4.5) that this parameter is used by mechanisms for reducing latency transfer.

It was mentioned in Section 2.4.1 that transmitted data can be corrupted if new data is written by the tracer in the sub-buffer while it is transferred. Sub-buffers that were corrupted can be detected by the examination of a status parameter that is transmitted at the end of each transfer.

Another solution for detecting such corrupted sub-buffers would be to make a copy of the whole sub-buffer just before the transmission begins, and then transfer this copy instead of the original data. The first solution was chosen as a first simplification in this work. Further studies will identify additional mechanisms.

*Table 1. Transferred tracefile header information.*

| Identifier | Type | Description |
|---|---|---|
| `header_size` | `uint32_t` | Size of the header in bytes. The size does not include the field `header_size`. |
| `magic_number` | `uint16_t` | Protocol identification number (this is a constant, the value must be equal to: `0x177d`). |
| `proto_version` | `uint16_t` | Protocol version (currently the value is: `0x0001`). |
| `subbuf_size` | `uint32_t` | Size for the sub-buffers of this tracefile (in bytes). |
| `tracefile_path` | `char[]` | Null-reminated string containing the relative tracefile path. |

[18] Integers are transferred using the *big-endian* convention.

*Table 2. Transferred sub-buffer information.*

| Identifier | Type | Description |
|---|---|---|
| `subbuf_bytes` | `uint32_t` | The number of bytes transmitted in `subbuf_data`. This value must be lower or equal to `subbuf_size`. |
| `subbuf_data` | `char[]` | Raw data contained in the sub-buffer. |
| `status` | `uint8_t` | The status of the transmitted sub-buffer. Possible values are:<br><br>-`SUBBUF_OK` (`0x00`):<br><br>-`SUBBUF_CORRUPTED` (`0x01`): |

**Implementation:**

This protocol was implemented directly in the *lttd* component as well as in a new *lttnetd* daemon (Figure 9 shows the modified version of *LTTng*). Referring to the client-server model used, the *lttnetd* daemon represents the server and *lttd* represents the client (Figure 2).

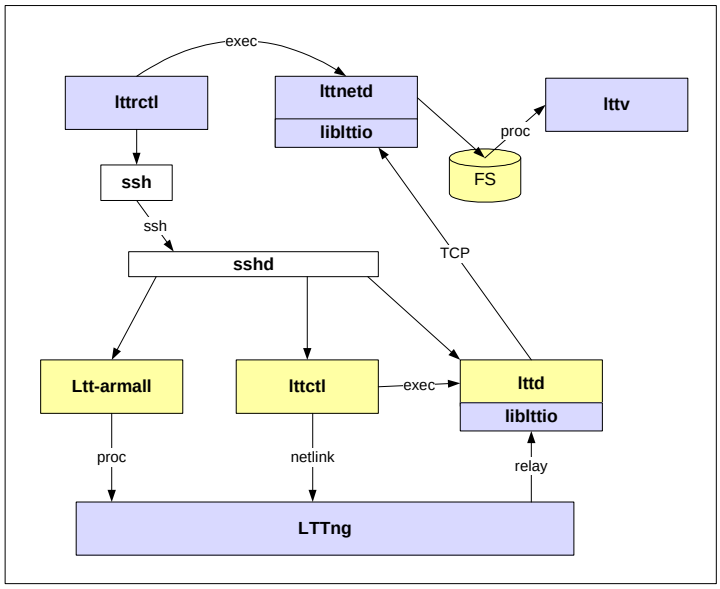


*Figure 9. Main components of LTTng – the Modified version*

The reception of execution traces by the server involves the use of the new API that is described in Section 2.4.3. The *lttctl* component was modified in order to expose the network transfer functionality (that is offered by *lttd*).

Our implementation manages multiple network connections concurrently (multiple concurrent distributed clients) with the help of one thread that loops with calls to the `pool(2)` function.

**Observations:**

It was shown in this task that the transfer of execution traces from one or many computational nodes to another one according to the client-server model is feasible with low technological risks. The use of TCP/IP protocol presents many advantages. It is a flexible, reliable and scalable mechanism that checks for transfer errors and manages flow control. TCP/IP flow may also be tunneled in an SSH tunnel (Figure 9). It can be used on many types of network links (wired, wireless, etc.) and many network topologies (LAN, MAN, WAN, etc.).

A number of improvements could be made to the system. First, currently *LTTng* has five channels (Section 2.4.1). If one multiplies this number by the number of apparent CPUs in the system and then multiplies the result by the number of distributed SMP or multi-core systems, it becomes apparent that connections that are concurrently sending execution traces toward the same server may cause network transmission performance problems. One solution envisioned would be to replace the traditional `pool(2)` function by a more efficient event notification mechanism. An example of such an improvement could be to use `epoll(7)` under Linux 2.6 or `kqueue(2)` under FreeBSD. Another solution would be to modify the communication protocol in order to multiplex into a single connexion all the tracefiles originating from a computational node.

Second, our implementation of *LTTng* is not multi-threaded; it does not make use of the parallelism capability that is offered on SMP or multi-core systems. Based upon CPU workload attribution, improvements in this direction may yield different strategies for monitoring, tracing and analyzing systems.

Finally, it should be mentioned that embedded systems may not have the necessary network infrastructure to support the proposed trace transfer protocol mechanism. In this case, our solution would need to be revised for this kind of systems.

### 2.4.5 Task 3 – Problem of latency transfer

As mentioned earlier, *LTTng* maintains buffers for each tracefile of every execution trace. These buffers are not flushed until they get full. This may present a problem in the context of online trace analysis; if the rate of event generation is not high enough, the data will stay in buffers for long periods of time, causing lags in trace analysis. These buffers must thus be regularly flushed.

**The Flushing of data buffers:**

The *LTTng* kernel component was modified in order to allow the flushing of data buffers on a regular basis. A timer was installed on each tracefile in order to trigger periodic flushing of the buffer. This timer is launched when the trace is started, and it is reset to zero when the buffer is flushed (either if the timer expires or the buffer gets full). The value of the timer can be specified

when the execution trace is created (by a `netlink` command). Specifying a negative value deactivates the timer.

**The Reduction of execution trace volume:**

Sub-buffers are stored entirely into the tracefile and they often include unused bytes at the end of the buffer. The premature flushing of the sub-buffer (triggered by ending timers) will cause the storage of partially filled sub-buffers and will cause significant memory waste. Ideally, sub-buffers should be of variable size so that only the bytes used would be stored. However, we are limited by the current version of *LTTV*, which only supports fixed size sub-buffers.

Modifications were made in order to optimize resource usage. First, an `ioctl` command was added to query (from user space) the number of used bytes in reserved sub-buffers. Second, the implementation of the trace transfer protocol was modified in order to transmit only the used bytes of the sub-buffers. Finally, the storage routine was changed to leave unallocated blocks inside the files, where there are irrelevant data regions. Unallocated blocks enable us to reduce storage consumption while keeping sub-buffers to a fixed size as required by *LTTV*.

**Observations:**

One timer was implemented per buffer. If we consider the possibility of having multiple concurrent CPUs, the number of timers may grow rapidly, and have negative performance impacts. A solution to this potential problem would be to use only one timer per CPU, and flush all channels at the same time.

Despite the fact that these optimizations contribute significantly to a reduction in wasted network traffic and storage space caused by premature buffer flushing, the mechanisms used are not perfect and should be improved. For instance, the unallocated blocks are not considered in the total file size and may confuse the user. Also, the optimization can be lost if tracefiles are moved to another system. An ideal solution would be to add variable sub-buffer size support into *LTTV*.

### 2.4.6 Task 4 – Remote control

This task aimed to investigate the possibility of achieving remote secure control of *LTTng*. SSH and its authentication mechanism were chosen for security and reliability reasons (Figure 9).

The prototyped solution implements the network transfer of commands for controlling *LTTng*. An API is used to encapsulate the interfacing with SSH and their sending over the network.

**Observations:**

Results of this prototyping effort show that the use of SSH significantly eases the implementation of the needed capability. There is no need to create a new control protocol, since this mechanism already offers authentication, encrypting and tunneling capabilities.

The effort and technological risks associated with this task are relatively low. One limitation of this solution is related to the use of SSH in embedded systems, where resources are restricted and networking support may be absent.

### 2.4.7 Task 5 – Decoding of received execution traces

Raw execution traces must be decoded and merged before they can be remotely analyzed on the server. The current implementation of *LTTV* cannot be used to achieve online remote analysis because this software only deals with complete local execution traces, which are already recorded on hard disks.

The feasibility of achieving remote online analysis of execution traces was shown. The modified version of *LTTV* (Figure 9) reads and decodes many traces online, and it works with many concurrent tracefiles. The development of the used prototype was not developed to its final stage because refactorization of *LTTV* is planned by the official programming team (LTTng, 2008).

**Observations:**

A mechanism for the synchronization of many concurrent tracefiles was implemented. One limitation of this mechanism is that it concurrently works exclusively in batch mode (for the moment). Technological risks associated with the management of received execution traces are relatively low.

Possible improvements to *LTTV* could be: 1) direct reading of execution traces from the network, without having to save the trace on the hard disk, and 2) continuous refreshing of the GUI showing received traces. These improvements should be done after the *LTTV* refactorization will be achieved by the official programming team.

The used prototype raises the problem of trace synchronization on the server side. Each trace element has a time stamp that was given by the clock of the computational node from which it originates. As the clocks of computational nodes are not synchronized, and they drift differently, the reassembling of trace elements that originate from many computational nodes will present a synchronization problem. This problem should be addressed as well.

# 3 R&D Thread 2 – Toward trace abstraction and analysis

The remote analysis of received execution traces on computational node 2 (Figure 3) was deliberately omitted in R&D Thread 1. R&D Thread 2 builds on the results obtained in R&D Thread 1 and evaluates the feasibility of remote trace abstraction and analysis. It aims to provide officers on duty with updated and accurate knowledge of their systems' health. Both R&D threads (when considered together as a whole) aim to evaluate the possibility of building **feedback-directed diagnostic systems**.

The main focus of this chapter is the study of theoretical and technical concepts and mechanisms allowing trace abstraction and analysis. The tracing software used in this chapter is the same as the one used in R&D Thread 1 (*LTTng*).

## 3.1 Initial premises and workload for this R&D thread

An **execution trace** is defined in this document as *a chronological suite of records resulting from observations that were made on a running system at different instants "$t_n$", during a specific period "delta-t"*. Trace events are captured when "*LTTng* probes" located in the software are encountered or executed. Depending on the number of active probes in the system, it is clear that execution traces may quickly become very huge, making element-by-element analysis nearly impossible to achieve.

The feasibility of trace abstraction and analysis is studied in R&D Thread 2. The strategy considered consists in: 1) the analysis of the inherent semantic of *LTTng* trace elements, 2) the identification of interrelationships between them, and 3) the building/use of execution models to deduce higher-level behaviour abstractions. The main premises that were considered in this work are:

1) The analysis of *LTTng* trace elements will aid in abstracting them into more significant (or higher level) behaviour abstractions.

2) Trace abstraction will in turn aid in improving many aspects of trace analysis and comparison.

3) Trace abstraction and analysis can be done on a continuous basis (during operations), thereby providing a continuous update of officers' knowledge regarding the health of their systems.

The workload of this R&D thread was structured in four different but complementary and sequential tasks. The goals of each task are briefly described below.

**Task 1 – Overview of current related works:**

Trace abstraction and analysis are relatively new domains in computer science. The main goal of this task consists in conducting a review of the scientific literature and identifying potential proven solutions that could aid in solving described problems.

**Task 2 –Analysis of execution traces:**

The second task in R&D Thread 2 consists in analyzing execution traces that were generated by *LTTng*. The goal consists in understanding their content, inherent logic and structure and then deducing hints for the feasibility of trace abstraction and analysis.

**Task 3 –Theoretical considerations for trace abstraction:**

Having understood *LTTng* execution traces, the next task consists in studying the feasibility of abstracting them into higher level behaviours for subsequent analysis.

**Task 4 – Program analysis toolkit (PAT) for automating trace analysis:**

This task is dedicated to the study of technical and practical aspects of the problem. It aims to verify the feasibility of automatic trace analysis during operations. This task will involve the development of prototypes that implement studied theoretical concepts and mechanisms.

## 3.2 Experiment

This section describes the experiment that was carried out to produce the studied execution trace. Figure 10 illustrates the system used, and the following lines provide a brief description of the experimentation.

Linux was used as the operating system. In this context, the CPU executes processes in one of the two execution modes: *user mode* in the **user space** or *supervisor mode* in the **kernel space**[19]. Interactions of user space applications with the hardware are usually achieved through calls to the *glibc* library, which make calls to appropriate **system calls**. System calls then trigger the execution of appropriate kernel modules in the kernel space. Returning notifications from the hardware to the user space application is usually achieved through "**interruptions**" (not illustrated in Figure 10 for clarity purposes).

In this experiment, the running application (user space; client side) is *micro_httpd*[20] (Micro_httpd, 2008). This application is a tiny Unix-based HTTP server (7 Kb). It runs from messages arriving from the daemon *xinetd* and it implements all the basic features of an HTTP server including security, filename snooping, the common MIME types, trailing-slash redirection, index.html, and directory listings.

Briefly, *xinetd* manages Internet services on Linux systems by listening on specified ports (80 in our case), and redirects requests to the appropriate server (*micro_httpd*). The server *micro_httpd* receives requests from *xinetd* and then offers the service for this request (sending back a web page for instance).

[19] More technical details on Linux and its kernel can be found in Corbet et al. (2005) and Fusco (2007).
[20] The source code of this application is listed in Annex D.

In this experiment, *LTTng* produces execution traces by capturing events that are triggered **at the user/kernel interface (system calls)**.

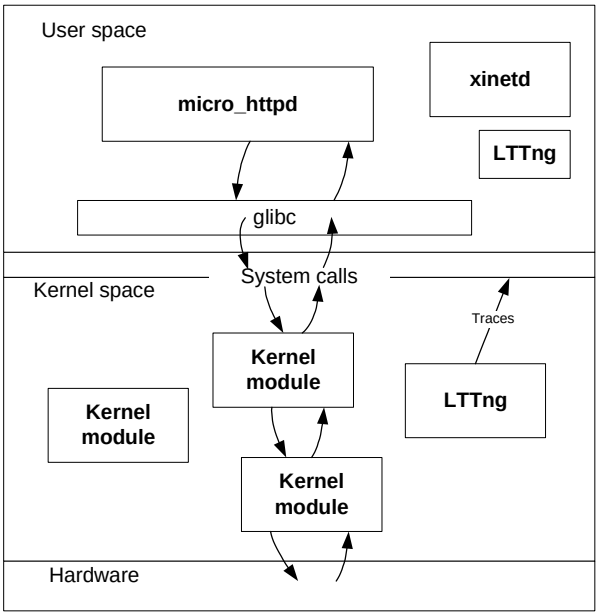


*Figure 10. Cascade effects of the execution of the micro_httpd application.*

All figures in this chapter show execution traces that are actually portions of one big execution trace, which is listed in Annex E. This execution trace corresponds to a full execution of the *micro_httpd* application. The reader may compare directly this execution trace with the source code of *micro_httpd* (Annex D.1). This will provide a complete picture of events that are happening at the kernel's interface, notably at the moment *micro_httpd* is started and stopped.

## 3.3 Exploration and analysis – Results and observations

### 3.3.1 Task 1 – Overview of current related works

The field of trace analysis is still in its infancy. To our knowledge, no trace analysis technique or technology can be used generically in different contexts. Currently, each technique addresses one or a few specific problems and is hard to adapt for solving different problems. However, there are a number of similar fields from which it is possible to draw inspiration for the design of trace analysis solutions. Of these fields, intrusion detection (ID) offers particular promise.

This section provides a quick overview of the current work that could make significant contributions to trace abstraction and analysis.

#### 3.3.1.1 Trace analysis techniques

Four categories of trace analysis can be identified. They are described below.

**Trace visualization:**

Trace visualization aims to make it easier to understand trace elements. It makes use of different graphic representations and related facilities (such as zooming) to show precisely selected parts of the execution trace. Two important references are Hamou-Lhadj (2005) and Lianjiang (2005).

Two examples of a visualization tool that can be used to identify similarities (or differences) between execution traces are: Cornelissen and Moonen (2007) and Fischer et al. (2005). The first paper aims to identify recurrent patterns in execution traces, while the second aims to follow the evolution (execution) of a program by the visual comparison of traces.

**Pattern detection in execution traces:**

Pattern detection in execution traces is somewhat similar to intrusion detection (ID), but it addresses a broader spectrum[21] than ID. Pattern detection can be used to verify program integrity (detecting bugs in a program) and to identify tasks that place heavy demands on the CPU.

Some selected papers on pattern detection in execution traces are: Ezust and Bochmann (1995): verifying that traces conform to specifications; Lau et al. (2003): learning program behaviours without access to the source code; Langevine (2002): constraint programs trace analysis; Chen et al. (2003): verifying system design through analysis of traces using constraint language (logic of constraint); and Paxton (1997): analyzing TCP protocol implementation through the analysis of exchanged packets.

**Statistics on trace elements:**

A number of measurements are made on the elements of a trace in order to produce statistical information regarding the execution of the program. Some examples of these statistics are: “the most often observed behaviour”, “the proportion in which a method or system call is called”, etc.

Chapter 3 of Hamou-Lhadj (2005) presents numerous metrics that can be used to study execution traces.

**Trace compression:**

Compression aims to reduce the size of execution traces by grouping together selected elements or parts. For instance, a method is proposed by Hamou-Lhadj (2005) to group many similar trace elements that originate from the repetitive execution of the same function within the program.

#### 3.3.1.2 Intrusion detection

The intrusion detection field has yielded a number of concepts and mechanisms for analysis that must be considered in trace analysis[22]. In this context, the term intrusion means “an attempt to get confidential information to maliciously exploit system resources”.

[21] As shown later in this section, ID mainly addresses issues of security and related behaviours.
[22] Couture, Mathieu (2005) provides a good description of Intrusion Detection (ID) systems, languages and analysis. A number of important references in this field are provided as well.

ID aims to detect: 1) if information confidentiality has been compromised and 2) malicious uses of system resources. In order to address the ID problem, three approaches were identified in the literature: behaviour, scenario and mix approaches. They are introduced below.

**Behaviour approach:**

The behaviour approach has two distinct phases. In the first phase, a model of the user's behaviour is built. In the second phase, an evaluation of the difference between this model and observation of other users' behaviours is made in order to detect anomalies that could originate from malicious activities on the system.

The building of the model is achieved through a learning process. No matter what techniques are used, the same problem must be addressed: when can we say that learning can be stopped? One solution that was considered in the literature involves learning on a continuous basis.

The main advantages of this approach are that it does not prescribe a predefined list of malicious behaviours, and this allows it to identify previously unknown malicious activities. This approach is often used to verify if users' behaviours are acceptable. A significant disadvantage of this approach is that it may trigger false positives.

**Scenario approach:**

Here, observations are compared with a pre-defined list of unwanted behaviours. The goal is to detect well known specific unwanted behaviours[23].

The main advantage of this approach is that it will not trigger false positives. A significant disadvantage is that not all unwanted behaviours are known in advance, and some important unwanted behaviours may be inadvertently missed.

**Mix approach:**

This approach involves the concurrent use of both behaviour and scenario approaches. The goal is to exploit advantages and minimize disadvantages. For instance, CISCO IDS (Cisco_IDS, 2008) makes use of this approach.

Table 3 lists the advantages and disadvantages of the behaviour and scenario approaches.

*Table 3. Advantages and disadvantages of described ID approaches.*

| | **Behaviour approach** | **Scenario approach** |
|---|---|---|
| **Advantages** | Possible detection of unkown unwanted behaviours | No false positives |
| **Disadvantages** | False positives | False negatives |

[23] Unlike the behaviour approach.

| | | |
|---|---|---|
| | False negatives<br><br>Relatively long learning process | Limited to well known scenarios |

#### 3.3.1.3 Kleene algebra

The Kleene algebra[24] (Lajeunesse-Robert, 2008; Dexter, 1990; Dexter, 1994; Dexter, 2004) is a mathematical tool used to represent software programs in abstract form. Actually, it represents a formalism containing a set of **rules** that capture the semantic of program operations, and then permits abstraction manipulation and significantly facilitates logical reasoning. Some types of software operations (instructions) that can be represented by this formalism are sequences of actions, choice between actions, and iteration of actions. The application of a set of rules to a software program will allow it to be progressively transformed into the form of a Kleene algebraic expression, which will serve as the basis for analysis, comparison, verification, etc. This algebraic expression formally represents an abstraction of the software program.

A number of studies have shown that by using these rules it is possible 1) to formally prove that two programs are equivalent (Dexter, 1997a and 1997b); 2) to certify that code optimization made by a compiler is correct (Dexter and Patron, 2000); 3) to verify that a program has expected behaviours (Bolduc, 2006; Dexter, 2003); 4) to prove the consistency of a communication protocol (Paxton, 1997); 5) to verify compiled java code (Kot and Kozen, 2005); and 6) to achieve model verification (Ktari et al., 2008). However, it must be said here that reasoning with this formalism may become very difficult to achieve in certain circumstances. In other words, despite the fact that it may be easy to transform a software program into an algebraic expression, it may be very hard (even impossible) to find how to prove the equivalence of two programs (Cohen et al., 1996; Hardin and Kozen, 2002; Hardin and Kozen, 2003; Hardin, 2005; Dexter and Patron, 2000).

Another significant advantage of the Kleene formalism is that it can take into account different realities or contexts. Examples of R&D efforts to adapt the Kleene algebra to software programs involving complex operations other than the ones listed earlier (such as C “pointers” and others) can be found in Kamal and Kozen (2007), Allegra and Kozen (2001), Adam and Kozen (2002), Desharnais et al. (2004), Desharnais et al. (2006), Ehm (2004), Jipsen (2004), Dexter (1990), Dexter (1997a and 1997b), Dexter (1998), Dexter (2004a and 2004b), Leiß (2006) and Mathieu (2006).

It must be borne in mind that, despite the lack of any evidence tending to limit the types of reality that can be expressed by this formalism, adapting it may require considerable effort.

[24] Studies such as Neumann (2000) and Neumann (2004) (and many others such as Schneider (1999) and Jackson et al. (2007)) recommend the consideration of formal methods for *demonstrating consistency of specification with requirements, and consistency of code with specification; formal verification and model checking; analysis tools for detecting characteristic security flaws, buffer overflows*, etc.

### 3.3.2 Task 2 – Analysis of execution traces

The semantic of trace elements, their inherent logic and interrelationships between them are studied in greater depth in this section.

**Content of *LTTng* execution traces:**

Figure 11 shows a portion of the execution trace that was generated by *LTTng* while executing *micro_httpd* (lines 35-39 of Annex E). Each line corresponds to the execution of a *LTTng* probe (herein called "event"), which are all located at the interface user/kernel space (system calls). The meaning of selected fields (bolded in line 35) is the following:

- **`kernel_arch_syscall_entry`**: *LTTng* marker ID (the probe identification).
- **`5277.041364003`**: the time when the event was recorded.
- **`4566`**: PID, or process identification.
- **1** (just before "`0x0, SYSCALL`"): PPID, or parent process identification.
- **`SYSCALL`**: it provides information regarding the mode of the execution (could also be USER_MODE like in line 39) as well as kernel's activities.
- **`{syscall_id = 120 [sys_clone+0x0/0x40], ip = 0xb8046424}`**: attributes associated with the event. For instance, in this case syscall_id = 120 corresponds to "clone" or create a child process.

Actually, this portion of the execution trace shows that the daemon *xinetd* receives a "GET /index.html" request. First, it carries out a `fork` (line 36) to create a new child process (which is the *micro_httpd* server). Second, it wakes up and schedules the new process (lines 37, 38) and then passes the control to the newly created process (line 39 is the end of the process creation) for the request to be serviced (not shown in this figure).

In this example, the *xinetd*'s process number is 4566, *xinetd*'s parent process is the well known UNIX/Linux *init* process (PID = 1), the newly created process has a PID equal to 4979 and process 4979's parent process is *xinetd* (PID = 4566).

This newly created process is not a thread; it does not share the resources of its parents. It is rather a regular process having its own resources. Lines 133-137, and more particularly line 136 of Figure 12, correspond to the launching of this process (`fs_exec`).

```
# xinetd carries on a fork(), the CPU is available for the process that was just created

35- kernel_arch_syscall_entry: 5277.041364003 (/tmp/trace-httpd/cpu_0), 4566, 4566,
xinetd, , 1, 0x0, SYSCALL { syscall_id = 120 [sys_clone+0x0/0x40], ip = 0xb8046424 }

36- kernel_process_fork: 5277.041457157 (/tmp/trace-httpd/control/processes_0), 4566,
4566, xinetd, , 1, 0x0, SYSCALL { parent_pid = 4566, child_pid = 4979, child_tgid = 4979 }

37- kernel_sched_wakeup_new_task: 5277.041459577 (/tmp/trace-httpd/cpu_0), 4566, 4566,
```

```
xinetd, , 1, 0x0, SYSCALL { pid = 4979, state = 0 }

38- kernel_sched_schedule: 5277.041466725 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, ,
4566, 0x0, SYSCALL { prev_pid = 4566, next_pid = 4979, prev_state = 0 }

39- kernel_arch_syscall_exit: 5277.041486672 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd,
, 4566, 0x0, USER_MODE { ret = 0 }
```

*Figure 11. Execution trace – Portion A.*

```
# Launch “micro_httpd”

131- kernel_arch_syscall_entry: 5277.045111538 (/tmp/trace-httpd/cpu_0), 4979, 4979,
xinetd, , 4566, 0x0, SYSCALL { syscall_id = 66 [sys_setsid+0x0/0xd0], ip = 0xb8046424 }

132- kernel_arch_syscall_exit: 5277.045115067 (/tmp/trace-httpd/cpu_0), 4979, 4979,
xinetd, , 4566, 0x0, USER_MODE { ret = 4979 }

133- kernel_arch_syscall_entry: 5277.045125131 (/tmp/trace-httpd/cpu_0), 4979, 4979,
xinetd, , 4566, 0x0, SYSCALL { syscall_id = 11 [sys_execve+0x0/0x80], ip = 0xb8046424 }

134- fs_close: 5277.045319969 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0,
SYSCALL { fd = 5 }

135- fs_close: 5277.045394854 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0,
SYSCALL { fd = 3 }

136- fs_exec: 5277.045421309 (/tmp/trace-httpd/control/processes_0), 4979, 4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { filename =
"/usr/local/sbin/micro_httpd" }

137- kernel_arch_syscall_exit: 5277.045424721 (/tmp/trace-httpd/cpu_0), 4979, 4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, USER_MODE { ret = 0 }
```

*Figure 12. Execution trace – Portion B.*

**Possible de-multiplexing of execution traces:**

As shown in this example, the *LTTng* framework provides information regarding current and newly created processes that are running in the system[25]. Knowing PIDs and PPIDs, it is relatively easy to segregate (or de-multiplex) the events of a raw trace based upon these identification numbers. The result of de-multiplexing is a number of sub-traces, each reflecting activities related to one specific process running in the system.

Interrelationships between processes and open files can be identified by searching execution traces for information such as:

[25] Similar information on running processes can be obtained with the use of other Linux commands and tools such as strace, ps, top, /proc/ GProf, etc.

- *LTTng* marker ID "`list_file_descriptor`". Taking for instance line number 12 (Figure 13), one can see that process 4974 is performing an operation on a file (file descriptor fd = 3). Line number 12 also shows that the file is a pipe[26].
- Operations on these files. Lines 23, 55, 62, 90, 97 and 101 (Figure 13) are good examples (`fs_select, fs_close, fs_open, fs_read`). Process PID's are included in each of these lines.

```
9- list_file_descriptor: 5270.945015874 (/tmp/trace-httpd/cpu_0), 4974, 4974, lttctl, ,
4973, 0x0, MODE_UNKNOWN { filename = "/dev/null", pid = 4566, fd = 0 }

10- list_file_descriptor: 5270.945017342 (/tmp/trace-httpd/cpu_0), 4974, 4974, lttctl, ,
4973, 0x0, MODE_UNKNOWN { filename = "/dev/null", pid = 4566, fd = 1 }

11- list_file_descriptor: 5270.945018858 (/tmp/trace-httpd/cpu_0), 4974, 4974, lttctl, ,
4973, 0x0, MODE_UNKNOWN { filename = "/dev/null", pid = 4566, fd = 2 }

12- list_file_descriptor: 5270.945021038 (/tmp/trace-httpd/cpu_0), 4974, 4974, lttctl, ,
4973, 0x0, MODE_UNKNOWN { filename = "pipe:[9851]", pid = 4566, fd = 3 }

13- list_file_descriptor: 5270.945022718 (/tmp/trace-httpd/cpu_0), 4974, 4974, lttctl, ,
4973, 0x0, MODE_UNKNOWN { filename = "pipe:[9851]", pid = 4566, fd = 4 }

14- list_file_descriptor: 5270.945024729 (/tmp/trace-httpd/cpu_0), 4974, 4974, lttctl, ,
4973, 0x0, MODE_UNKNOWN { filename = "socket:[9860]", pid = 4566, fd = 5 }

15- list_file_descriptor: 5270.945026921 (/tmp/trace-httpd/cpu_0), 4974, 4974, lttctl, ,
4973, 0x0, MODE_UNKNOWN { filename = "socket:[9854]", pid = 4566, fd = 7 }

23- fs_select: 5277.041308682 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0,
SYSCALL { fd = 3, timeout = -1 }

55- fs_close: 5277.041884668 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0,
SYSCALL { fd = 3 }

62- fs_close: 5277.041891919 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0,
SYSCALL { fd = 0 }

90- fs_open: 5277.044684136 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0,
SYSCALL { fd = 0, filename = "/etc/hosts.allow" }

97- fs_read: 5277.044759137 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0,
SYSCALL { fd = 0, count = 4096 }

101- fs_close: 5277.044946441 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0,
SYSCALL { fd = 0 }
```

*Figure 13. Execution trace – Portion C.*

The utilization of system resources by executing programs will trigger typical sequences of events at the user/kernel interface (system calls). These events will be captured by the *LTTng* framework and saved in an execution trace. Specific sequences of trace elements could be recognized as behaviours related to resource utilization. The suite of lines 90, 97 and 101 in Figure 13 is an

[26] Recall that for Linux (and UNIX-like) operating systems everything (including pipes) is considered as a file.

obvious example showing such a sequence of events for the utilization of the file `hosts.allow` by one process.

The identification of such sequences in execution traces may be problematic if accesses to the file are achieved by other concurrent processes. For instance, this situation happens when a file that was created by a process is accessed by one of its child processes[27]. The inclusion of sub-tasks within tasks may also make it harder to identify sequences in execution traces. Following are some conditions that make it easier to identify interrelationships between executions of concurrent processes.

1. Keeping an up-to-date table showing correspondences between files and processes. Knowing the parent-child relationships between active processes, it is possible to deduce to which parent the process that accessed the file belongs to.

2. For each system call that is called by the user's application, two *LTTng* markers are recorded in the trace: "`kernel_arch_syscall_entry`" and "`kernel_arch_syscall_exit`". One should verify that trace elements lying in between these markers are exclusively related to the called system call.

3. Levels of these two *LTTng* markers should be taken into account as well. Figure 14 shows an example of inclusion having two hierarchical levels; level one (lines 271 to 281) contains level 2 (lines 274 to 279).

The following important observations can be made.

- It is possible to de-multiplex raw execution traces in sub-traces based upon process identification numbers.
- Considering the information available in execution traces, it is relatively easy to establish interrelationships between processes, and keep up-to-date the whole hierarchical structure of processes.
- It is possible to establish dependencies between processes and files, and identify operations that were made on these files.
- It is possible to identify the files that are shared among processes.
- These analyses may possibly be done automatically.

```
271-  kernel_arch_syscall_entry:  5277.047178396  (/tmp/trace-httpd/cpu_0),  4979,  4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { syscall_id = 4 [sys_write+0x0/0xa0],
ip = 0xb8046424 }

272-     fs_write:     5277.047179914     (/tmp/trace-httpd/cpu_0),     4979,     4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd = 1, count = 311 }

273-    net_dev_xmit:    5277.047200820    (/tmp/trace-httpd/cpu_0),    4979,    4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { skb = 0xd96c9cb4, protocol = 8 }

274-   kernel_softirq_entry:   5277.047206305   (/tmp/trace-httpd/cpu_0),   4979,   4979,
```

[27] For Linux, a parent process may create child processes. Under certain conditions (when the child takes the form of a thread), both processes (parent and child) may share the parent's resource.

```
/usr/local/sbin/micro_httpd,    ,    4566,    0x0,    SOFTIRQ    {    softirq_id    =    3
[net_rx_action+0x0/0x240] }

275-    net_dev_receive:    5277.047209489    (/tmp/trace-httpd/cpu_0),    4979,    4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SOFTIRQ { skb = 0xd96c9cb4, protocol = 8 }

276-     net_dev_xmit:     5277.047224538     (/tmp/trace-httpd/cpu_0),     4979,     4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SOFTIRQ { skb = 0xd96ade00, protocol = 8 }

277-   kernel_sched_try_wakeup:   5277.047228132   (/tmp/trace-httpd/cpu_0),   4979,   4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SOFTIRQ { pid = 3689, state = 1 }

278-    net_dev_receive:    5277.047235954    (/tmp/trace-httpd/cpu_0),    4979,    4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SOFTIRQ { skb = 0xd96ade00, protocol = 8 }

279-    kernel_softirq_exit:    5277.047238183    (/tmp/trace-httpd/cpu_0),    4979,    4979,
/usr/local/sbin/micro_httpd,    ,    4566,    0x0,    SYSCALL    {    softirq_id    =    3
[net_rx_action+0x0/0x240] }

280-    kernel_timer_set:    5277.047241141    (/tmp/trace-httpd/cpu_0),    4979,    4979,
/usr/local/sbin/micro_httpd,  ,  4566,  0x0,  SYSCALL  {  expires  =  1486724,  function  =
0xc03336b0, data = 3745083136 }

281-   kernel_arch_syscall_exit:   5277.047252314   (/tmp/trace-httpd/cpu_0),   4979,   4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, USER_MODE { ret = 311 }
```

*Figure 14. Execution trace – Portion D.*

**Running processes – Trace analysis:**

Once the execution trace has been de-multiplexed into sub-traces (each of them corresponding to the execution of a specific process), it is easier to analyze process activities or behaviours. A first criterion that can be used to discriminate trace elements of sub-traces is the mode under which the processes are executed. It is possible to identify the mode by searching execution traces for markers starting with the labels "`_entry`" and "`_exit`". The mode corresponds to the type of the marker "`_entry`".

As shown in the last example (Figure 14), the levels of "`_entry`" and "`_exit`" markers are easily identifiable. The first level of abstraction ("send an ACK") corresponds to lines 271-281 and the second level ("deal with interruptions") corresponds to lines 274-279. It should be noted that problematic situations could happen if "`_entry`" markers are not closed by their corresponding "`_exit`" markers.

Many levels of abstraction can be identified if we consider: 1) the levels of these two markers; and 2) if deeper events within the kernel are considered. The activity of the kernel can also be deduced by examining the marker specifying the mode of the kernel. For instance, line 274 in Figure 14 tells that the kernel is dealing with interruptions (`kernel_softirq_`). Possible values of this field should be identified and described as well in future studies.

**Trace abstraction – Preliminary technical considerations:**

It was shown in this section that it is possible to de-multiplex execution traces into sub-traces, each corresponding to a running process. It is also possible to group trace elements of sub-traces

according to their inherent semantic. We will now study how execution traces can be abstracted into higher-level behaviours.

The process that is considered for trace abstraction is based on the building and use of an execution model that represents the user's software application. This model reflects all the effects that this program may trigger in the system (in our case, at the level of system calls). The model maps all program functions (high-level behaviours) with events that are expected to happen at the user space/kernel space interface when they are executed. There can be one or many levels of abstraction in the model.

The availability (or not) of the source code of both the user's program and the operating system (OS) is a determining factor in the execution model building process. Both situations lead to quite different building methods. In the case where the **source code is not available**, a learning phase involving the running and tracing of the user's application (and OS) in numerous different situations is necessary to generate (or deduce) the model. It is generated by compiling the analysis of these numerous execution traces. Not having the source code presents a number of limitations for trace abstraction and analysis. This case is not considered in this study.

In the case where the **source code is available**[28], the learning phase is replaced by a static (and dynamic) analysis of the source code and compiled programs. The execution model has a greater chance of being more complete and precise in this case. Nevertheless, trace abstraction and analysis will always be limited by the content of execution traces. It is impossible to trace all aspects of the kernel with full resolution; the size of the trace would be unmanageable. Traces will thus always contain a limited number of elements.

The utilization of the execution model (when the corresponding software program is executed) is explained here. Figure 15 is a theoretical representation of relations that may exist between: 1) traced events of an executing program, and 2) paths of an execution model. As the program evolves over time, different functions (potentially processes) of the program are executed. These functions trigger system calls, and they generate new events that are captured in execution traces by the *LTTng* framework. These events are then compared with events in each execution path of the model in order to identify the functions that were executed. This verification process is similar to that of "string matching".

Missing events[29] in execution traces can make this identification process more complex. An incomplete trace may not allow the identification of a precise path in the model. Moreover, low-resolution execution traces may cause problems if similar processes are executed simultaneously. This strongly suggests that an effort should be made to ensure that the level of detail within execution traces (or the number of active *LTTng* probes) is sufficient for the analysis needed, and there is no loss of elements during transmissions between computational nodes.

Another potential problem is the possibility that execution models will not have a sufficient level of detail. This problem would necessarily lead to situations in which specific behaviours would be

[28] The choice of using FOSS was made in this project (Section 2.2.1).
[29] Recall that execution traces cannot be comprehensive; traces would be unmanageable. Consequently, there will always be missing events in execution traces.

inadvertently missed. It may happen, for instance, if an execution trace contained information regarding interruptions in the kernel that are not taken into account in the model.

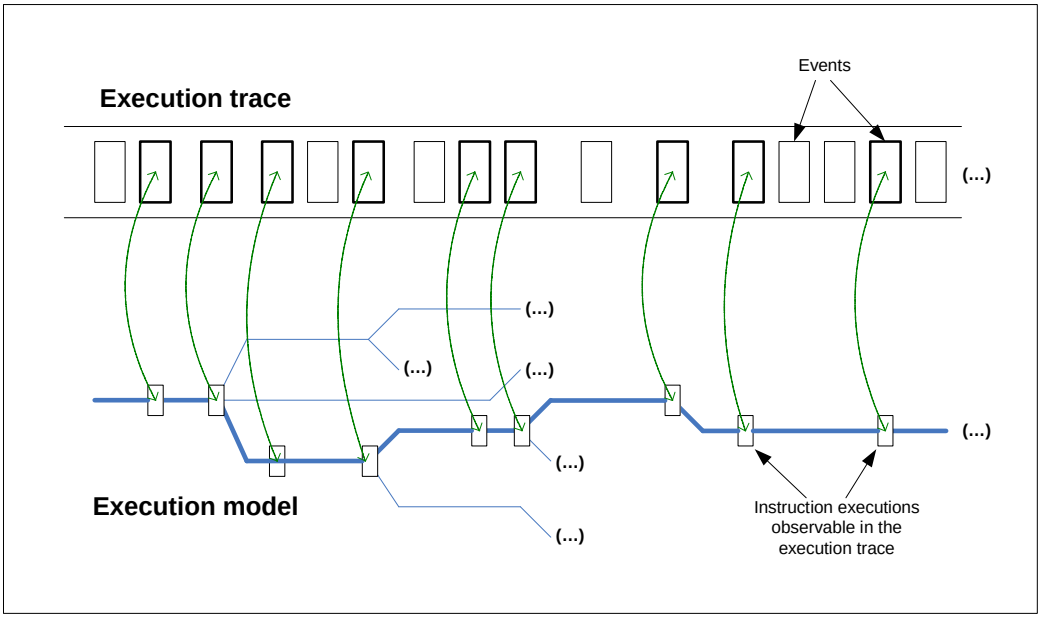


*Figure 15. Correspondence between an execution trace and execution model.*

The chosen approach for studying the abstraction process is divided into two parts: 1) the study of behaviours that are not exclusive to specific users' programs, and 2) the study of behaviours that are exclusive to specific users' programs.

### Case 1 – Behaviours not exclusive to specific users' programs:

Suites of related events in traces (common behaviours such as the opening, reading and closing of files) that may be triggered by many different processes in the user space should first be identified. Most of the time, their events will be scattered in the execution trace, and not necessarily ordered. Efforts should be made to capture these suites of events and the logic of their chronology (if any) in an execution model.

Deeper activities in the kernel may trigger activities independently of the users' processes as well. These events are basic operations achieved by kernel modules to keep the whole system operational and optimized with respect to time and activities. They may be visible in traces if deep *LTTng* probes are activated in the kernel. Efforts should be made to capture these suites of events and the logic of their chronology (if any) in an execution model.

### Case 2 – Behaviours that are specific to users' programs:

Behaviours that are specific to users' programs must be captured in the form of execution paths in the execution model. As mentioned earlier, the identification of the running process will involve the comparison of traced elements with those of the execution model. Once a minimum number

of trace elements are recognized in one of the execution paths of the model, the function or process will be identified (with a certain level of certitude).

The following example refers to Figure 16. It shows the mapping between two portions of the source code of the *micro_httpd* server with the two corresponding portions of the execution trace. The first mapping involves the function `chdir()` and the second involves the reading of the time system. Call to functions `chdir()` and `time()` triggers system calls that can be observed in the execution trace.

In this case, the limited resolution[30] of the trace would make its content harder to interpret. There is thus a need to be able to control the tracing software in such a way that it would be possible to produce high resolution traces with appropriate focus when needed. The choice of specific resolution and focus would depend on the type of abstraction and analysis that would be achieved.

**Mapping #1**:

**Lines of code:**

```
59- if ( chdir( argv[1] ) < 0 )

60- send_error( 500, "Internal Error", (char*) 0, "Config error - couldn't chdir()." );
```

**Execution trace elements:**

```
182- # “micro_httpd” carries on a chdir() in order to move into the directory containing
server’s files

183- kernel_arch_syscall_entry: 5277.046274494 (/tmp/trace-httpd/cpu_0), 4979, 4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { syscall_id = 12 [sys_chdir+0x0/0x70],
ip = 0xb8046424 }

184- kernel_arch_syscall_exit: 5277.046283568 (/tmp/trace-httpd/cpu_0), 4979, 4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, USER_MODE { ret = 0 }
```

**Mapping #2**:

**Lines of code (in the function `send_header()`):**

```
147- now = time( (time_t*) 0 );
```

**Execution trace elements:**

```
224- #Read the system time (to be inserted in the HTTP header; see send_header())

225- # (glibc opens "/etc/localtime" to know the system’s time zone)

226- kernel_arch_syscall_entry: 5277.046854659 (/tmp/trace-httpd/cpu_0), 4979, 4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { syscall_id = 13 [sys_time+0x0/0x30],
```

[30] **Focus** refers to the specific software objects being traced and **resolution** refers to the number of active probes used.

```
ip = 0xb8046424 }

227-  kernel_arch_syscall_exit:  5277.046856255  (/tmp/trace-httpd/cpu_0),  4979,  4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, USER_MODE { ret = 1211549623 }

228-  kernel_arch_syscall_entry:  5277.046888105  (/tmp/trace-httpd/cpu_0),  4979,  4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { syscall_id = 5 [sys_open+0x0/0x40], ip
= 0xb8046424 }

229-      fs_open:      5277.046899066      (/tmp/trace-httpd/cpu_0),      4979,      4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd = 4, filename = "/etc/localtime" }

230-  kernel_arch_syscall_exit:  5277.046900144  (/tmp/trace-httpd/cpu_0),  4979,  4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, USER_MODE { ret = 4 }

237-  kernel_arch_syscall_entry:  5277.046925133  (/tmp/trace-httpd/cpu_0),  4979,  4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { syscall_id = 3 [sys_read+0x0/0xa0], ip
= 0xb8046424 }

238-      fs_read:      5277.046926752      (/tmp/trace-httpd/cpu_0),      4979,      4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd = 4, count = 4096 }

239-  kernel_arch_syscall_exit:  5277.046964889  (/tmp/trace-httpd/cpu_0),  4979,  4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, USER_MODE { ret = 3477 }

240-  kernel_arch_syscall_entry:  5277.046988062  (/tmp/trace-httpd/cpu_0),  4979,  4979,
/usr/local/sbin/micro_httpd,  ,  4566,  0x0,  SYSCALL  {  syscall_id  =  140
[sys_llseek+0x0/0xd0], ip = 0xb8046424 }

241-      fs_llseek:      5277.046990756      (/tmp/trace-httpd/cpu_0),      4979,      4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd = 4, offset = 3453, origin = 1 }

242-  kernel_arch_syscall_exit:  5277.046991972  (/tmp/trace-httpd/cpu_0),  4979,  4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, USER_MODE { ret = 0 }

243-  kernel_arch_syscall_entry:  5277.047000434  (/tmp/trace-httpd/cpu_0),  4979,  4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { syscall_id = 3 [sys_read+0x0/0xa0], ip
= 0xb8046424 }

244-      fs_read:      5277.047001351      (/tmp/trace-httpd/cpu_0),      4979,      4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd = 4, count = 4096 }

245-  kernel_arch_syscall_exit:  5277.047004344  (/tmp/trace-httpd/cpu_0),  4979,  4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, USER_MODE { ret = 24 }

246-  kernel_arch_syscall_entry:  5277.047008992  (/tmp/trace-httpd/cpu_0),  4979,  4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { syscall_id = 6 [sys_close+0x0/0x110],
ip = 0xb8046424 }

247-      fs_close:      5277.047010329      (/tmp/trace-httpd/cpu_0),      4979,      4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd = 4 }

248-  kernel_arch_syscall_exit:  5277.047015507  (/tmp/trace-httpd/cpu_0),  4979,  4979,
/usr/local/sbin/micro_httpd, , 4566, 0x0, USER_MODE { ret = 0 }
```

*Figure 16. Execution trace – Portion E.*

The capability of online modifying both the focus and the resolution of *LTTng* execution traces according to the analysis that is needed appears to be essential for trace abstraction and analysis.

The resolution of the model should be fixed and it should allow the capture of all behaviours that may be present in execution traces.

**Observations:**

So far, it was shown that it is possible to de-multiplex elements of execution traces produced by the *LTTng* framework according to the running processes in the Linux kernel. This de-multiplexing facilitates the analysis process. However, the trace abstraction appears to be possible only under specific circumstances or conditions.

- First, the source code of the traced application must be available if we are to be able to produce detailed and precise execution models that will be used in trace abstraction and analysis.
- Second, the resolution of the execution trace should be high enough to allow abstraction and the resolving of behaviours that are found in the execution model.
- Third, the execution model of the executed program must be sufficiently complete and precise.

A lack of resolution in execution traces may cause events to be associated with many different but similar execution paths in the model. In this case, execution paths may not be resolvable. For instance, it might be impossible to tell which of two similar functions was executed based on the comparison of the trace and model.

On the other hand, very high resolution traces may lead to the creation of enormous traces, so large that they would be impossible to manage. The appropriate selection of active probes in the system must thus be guided or driven by well studied and understood strategies. Strategies that can be used in operations remain to be identified and studied in future.

An interesting solution for the problem of inadequate resolution in execution traces could be solved by giving unresolved execution paths “probabilities” of being “the one”. For instance, a low resolution trace may not allow the resolution of two similar execution paths in the model; there are not enough significant elements in the trace. The probability of each execution path of being the one could be calculated based on the number of successful comparisons between trace elements and the model. The value of these probabilities will evolve over time and active probes. With the arrival of new events in the trace, evolution of the trend would be observed. It could eventually become possible to state with certitude which execution path was involved in the production of trace elements, even if the resolution of the trace was relatively low.

Trace resolution and the level of detail of the execution model are decisive for the identification of problematic behaviours in the system (malicious or not). The problem of detecting unwanted behaviours in systems can be tackled using two different and complementary strategies (Section 3.3.1).

- **Problematic behaviours are known**. This is similar to the virus detection process on PCs; a number of virus signatures are known and the detection process examines executable codes (and the execution model) in order to detect the presence of unwanted signatures. In our case, known problematic behaviours should be represented in the form of suites of

events or signatures (trace abstractions). Pattern recognition techniques such as "string matching" will then help to identify these behaviours in execution traces.

- **Problematic behaviours are unknown**. This case is more complex, as signatures of problematic behaviours are not available at runtime, and we know that their probability of occurrence in the system is not negligible. A strategy based upon system coherence verification could be adopted in this case. In our case, the system's coherence would involve many of its inherent metrics, their normal ranges and the interrelationships between them. They would complement the model of execution with parameters defining the "normal conditions" of the system.

Some critical events may regularly occur at specific times or situations in the system. The identification of these times and situations should be captured in an execution model. When the user's program is executed, we know (from its execution model) what critical events should be triggered and when. The presence of one or a combination of these critical events at the wrong time would raise a flag. The resolution and focus of *LTTng* active probes could be changed in order to produce more relevant execution traces. Based on analysis, appropriate actions would be taken on specific components of the system.

Some selected unsolved problems in this R&D thread are: the building of complex execution models; processes that share the same resource (chunks of memory, etc.); parallel processes on multi-core CPUs; finding the appropriate level of detail for an execution model; how many active *LTTng* probes are needed; the effect in execution traces of deliberately introduced errors in the system; the use of *LTTng* with Linux standardized tools.

### 3.3.3 Task 3 – Theoretical considerations for trace abstraction

This section presents results that were obtained while verifying the feasibility of trace abstraction and analysis. It presents theoretical and practical perspectives on trace abstraction and analysis.

**Initial simplifications and context:**

A number of simplifications were initially brought to this work in order to keep its finality achievable. They are:

- The first simplification was already mentioned at the beginning of this chapter: only system calls to the kernel are considered. Moreover, the value of their parameters was not considered. Parameters that are associated with each event are probably important[31]. They could for instance allow the identification of unwanted behaviours such as those that violate information integrity.
- Processes are studied individually. Nowadays, operating systems such as Linux concurrently execute many processes "at the same time". These processes are often interlinked (and not necessarily in a parent-child relationship). The analysis of many concurrent processes is left to future studies.
- Recursive functions are not considered in this work, and the execution of any function of the program is always considered as "finite".

[31] R&D work is needed in this direction to identify how they should be taken into account.

- It is assumed that we have access to the source code of all software, as well as the operating system.
- It is also assumed that static and dynamic analyses of programs have yielded a complete and detailed execution model, which defines an execution tree containing all possible execution paths.

Using these simplifications, the mapping of any portion of the source code with the corresponding portions of execution traces should be relatively easy to identify. For instance, comparing trace elements with the content of the program's execution model would allow the identification of one or many potential execution paths. Trace elements corresponding to the execution of a specific function could then eventually be abstracted by the name of this function.

**Generation of the program's execution model:**

The execution model of *micro_httpd* is constructed by capturing its behaviours, which in our case involves calls to system calls exclusively. More precisely, for each *micro_httpd* function that leads to a system call, an execution path is added to the execution model; it has the form of a directed graph. Annex D.2 shows the resulting model using the C programming language.

Under listed simplifications, the execution model can be represented by an algebraic expression by using the Kleene formalism[32]. This formal representation of the program allows logical reasoning regarding its behaviour and facilitates the identification of operations that will lead to trace abstractions.

Table 4 shows the algabraic expression of the *micro_httpd* execution model. Following is a description of how this model was obtained.

It was shown earlier that the Kleene algebra uses rules to capture the semantic of the basic program's operations. These basic operations are designated by what is called "**operators**". Examples are:

- A sequence of two consecutive programs is expressed by the operator "`;`". The sequence of "`fn_1`" followed by "`fn_2`" would thus be expressed as: "`fn_1; fn_2`".
- The choice between two programs is represented by the operator "`+`". The C instruction "`if`" determining the choice between two execution paths "`c_1`" and "`c_2`" would be expressed as: "`c_1 + c_2`".
- The finite number of iterations of a suite of instructions such as the while loop "`while … {P}`" is represented by the operator "`*`". This loop would be expressed as: "`P*`", meaning that the program "`P`" may be executed a finite number of times (including 0 times).
- Additionally, Kleene algebra represents programs that do nothing by "`1`" and programs that abort by "`0`".

[32] In the case where the nature of captured events was considered (the parameters of each System call), a modified version of the Kleene formalism (or another) should be used.

Using this Kleene representation scheme, it becomes easy to understand how the *micro_httpd* model can be translated into an algebraic expression. The reader will note that instructions "`if`" in the program do not have the associated "`else`" instruction, but the algebraic expression still considers the "`else`" by using the "`1`" operator.

*Table 4. Transcription of the micro_httpd model into a Kleene algebraic expression.*

| **Function name of *micro_httpd*** | **Kleene expression (program's model)** |
|---|---|
| `main()` | *(send error + 1); chdir; (send error + 1); fgets; (send error + 1); (send error + 1); Fgets*; (send error + 1); (send error + 1); (send error + 1); stat; (send error + 1); ((send error + 1); stat; (fopen; (1+send error); send headers; (getc; putchar)*; fflush; exit + send headers; scandir)+fopen; (1 + send error); send_headers; (getc; putchar)*; fflush; exit)* |
| `send_error()` | *send headers;*<br>*fflush;*<br>*exit* |
| `send_header()` | *time* |

**Trace abstraction:**

A number of system calls are done when *micro_httpd* is executed. They are captured in the execution trace. Using the execution model of *micro_httpd* that was just generated, it becomes relatively easy to identify the portions of the source code (execution paths in the model) that correspond to each element included in the execution trace.

Two cases will be discussed in this section. They are defined by the coverage of the execution trace.

- Case 1: **the execution trace covers the whole execution of the program**.
- Case 2: **only a portion of the execution trace is available**.

For each case, we will formally present what is possible to achieve in terms of trace abstraction and we will present how it can be achieved using the Kleene algebra.

The reader should note at this point that, for clarity purposes, we will use a theoretical program that is simpler than *micro_httpd.* It should be borne in mind that achieving the same type of analysis with the same simplifications (listed at the beginning of this section) for the *micro_httpd* program would have led to similar results regarding the feasibility of trace abstraction.

The model of the program that will be studied in this section is illustrated in Figure 17. The lines represent execution paths. This program is made up of three parts:

- part 1 (Fn1); it is executed after the event "e1" was triggered;

- part 2 (Fn2); it is executed after the event “e2” was triggered; and
- part 3 (Fn3); it is executed after the event “e3” was triggered.

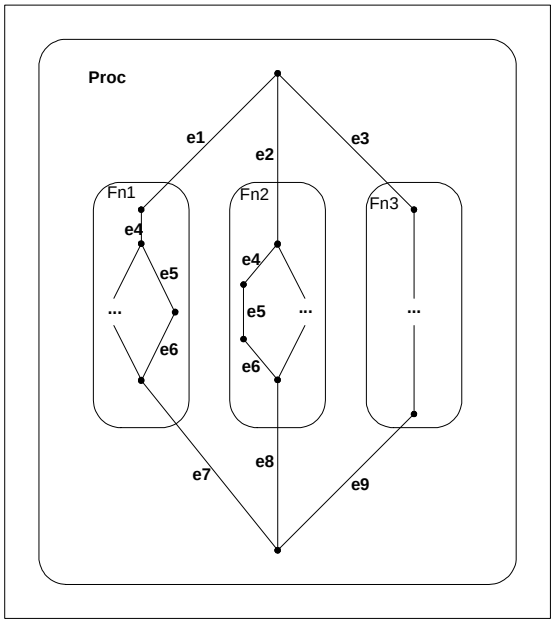


*Figure 17. Example of a program model (A).*

## Case 1 – The execution covers the whole execution of the program:

### a) Informal description:

One thing that can be done in Case 1 is to use the program’s execution model to check if captured events in the trace are generated by the program (and LTTng probes[33]). For instance, the program’s execution model illustrated in Figure 17 shows that the observation of events e1, e4, e5, e6 and e7 in the trace would correspond to one of the possible execution paths of the program’s execution model (the red execution path in Figure 18). An execution of Fn1 that would not yield these elements (e1, e4, e5, e6 and e7) would indicate that the executed program was modified in one way or another.

In Case 1, this process is made easier because the beginning of the execution trace corresponds to the beginning of the program execution. It is thus easy to “follow” (in the execution model) the execution of the program element by element, and identify executed paths. In other words, when the application is launched, we use events in the trace to identify where (in the model) the executed program stands. As the execution progresses, new events are compared with the model, allowing the identification of the new execution paths that are used.

[33] Recall that a process number is associated with each element (event) in the LTTng execution trace.

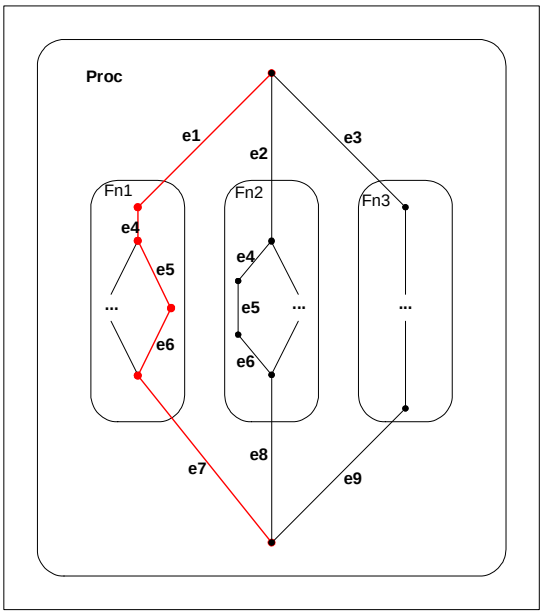


*Figure 18. Example of a program model (B).*

If the resolution of the execution trace is not high enough[34], trace elements may not be sufficient to identify the executed path. In this case, more than one execution path could then be associated with trace elements[35]. Nevertheless, further additional events in the trace would contribute to the identification of the execution path used, and eliminate the others.

The analysis of trace elements results in the identification of the execution paths. Knowing which function is currently executing, it is possible to abstract sequences of trace elements by the name of the function that was executed.

Referring to our example (Figure 18), we see that the execution trace could be abstracted by: “e1” followed by “Fn1” and “e7” (as illustrated in Figure 19 by red lines).

[34] There are not enough active probes in both the software and the operating system. The content of execution traces is not dense enough to allow the identification of execution paths.
[35] The end of Section 3.3.2 introduces the possibility of giving each found execution path a probability level.

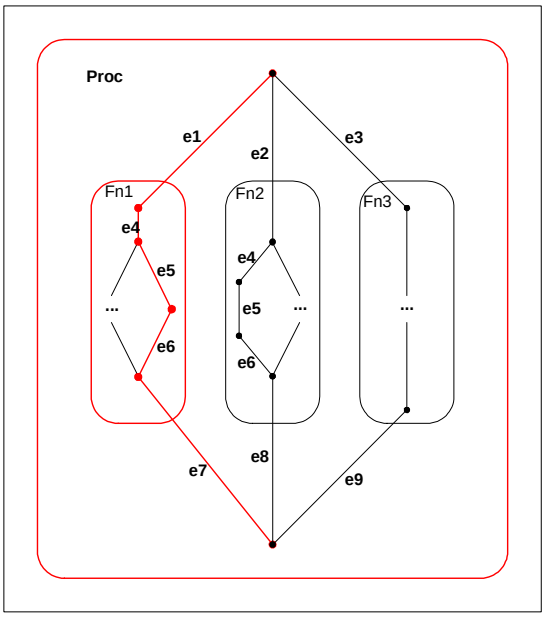


*Figure 19. Example of a program model (C).*

In real life, there would be additional events in the trace that would not necessarily be related to the execution of function Fn1. These events are caused by the execution of other programs or kernel modules[36].

**b) Using the Kleene algebra formalism:**

The Kleene formalism can be used to represent: 1) operations in execution traces and 2) the model of the executed program. It provides a means for formal reasoning. So far, we have shown a simple example of trace abstraction (Case 1). We will now present how the Kleene formalism can be used in this context. We will translate the execution model (Figure 17) into an algebraic expression.

Table 5 shows the transcription of our theoretical program (Figure 17) into an algebraic expression. The methodology and symbolism described in the previous section were used. The `main()` function is represented by a choice between the execution of three functions (Fn1, Fn2 and Fn3) and the symbol "…" designates portions of the execution model that are not given explicitly.

The execution model of a program lists all possible execution paths that can be executed while the program is running. To determine if an execution trace matches the execution model (expected behaviours), one must verify whether or not the execution trace is consistent with the model. Using the Kleene formalism, this can be expressed as:

$$t \leq M \quad \text{(Eq. 1)}$$

[36] But as shown earlier, we can discriminate them.

where “t” represents the observed execution trace and “M” the execution model. The methodology defined by Lajeunesse-Robert (2008) can be used to verify whether or not this inequality (Eq. 1) is valid. In our example (Figure 17), Eq. 1 is equivalent to the following algebraic expression:

$$e1; e4; e5; e6; e7 \leq main \qquad \text{(Eq. 2)}$$

*Table 5. Transcription of micro-httpd functions into Kleene algebraic expressions.*

| Function name | Kleene algebra expression |
|---|---|
| main() | e1; Fn1; e7 + e2; Fn2; e8 + e3; Fn3; e9 |
| Fn1 | e4; (... + e5; e6) |
| Fn2 | e4; e5; e6 + ... |
| Fn3 | ... |

The online study of program execution involves the identification of all execution paths in the model that contain captured trace elements (and the exclusion of all other paths). In other words, every time a trace element is captured a new execution model is generated, which keeps only execution paths that contain trace elements. The generation of this new model is easily represented by Eq. 3:

$$e \setminus M \quad \text{(Eq. 3)}$$

where “e” represents the observed event (trace elements) and “M” represents the current model. In our example (Figure 17), the capture of the trace element “e1” would yield a new execution model that would contain the set of possible execution paths. It can be represented by the following expression:

$$e1 \setminus main \qquad \text{(Eq. 4)}$$

Eq. 4 represents the execution model found in Table 6. The graphic representation of the execution model (Figure 17) shows that there is only one path that starts with trace element “e1”: the one that leads to the function “Fn1”. Keeping only the execution paths of the model that start with “e1” we obtain the algebraic expression listed in Table 6.

As new trace elements are captured, new execution models are generated from the current models. In the case where a trace element would not be included in the model’s logic, the result would be represented by:

$$e \setminus M \textbf{ is } \mathbf{0} \qquad \text{(Eq. 5)}$$

Having found the execution model that corresponds to the last execution of the program, we can then abstract the execution trace by the name of the functions that were executed. To do that, we must modify the execution model that was found in such a way that it will allow the inclusion of the information regarding the “range” of executed functions. Tags are added to the execution

model to designate the beginning and the end of each function. Table 7 illustrates this execution model in the form of an algebraic expression.

*Table 6. Reduced Kleene algebraic expression.*

| Function name | Kleene algebra expression |
|---|---|
| main | Fn1; e7 |
| Fn1 | e4; (... + e5; e6) |

*Table 7. Kleene algebraic expression with "tags" added.*

| Function name | Kleene algebra expression |
|---|---|
| main | main_begin; (e1; Fn1; e7 + e2; Fn2; e8 + e3; Fn3; e9); main_end |
| Fn1 | Fn1_begin; e4; (... + e5; e6); Fn1_end |
| Fn2 | Fn2_begin; (e4; e5; e6 + ...); Fn2_end |
| Fn3 | Fn3_begin; ...; Fn3_end |

The execution model found contains more information than real executions (the tags). These tags make it impossible to compare the found model with the execution trace because the latter does not contain tags. To solve this problem, a dummy event is added between each element of the trace in order to specify that tags may be encountered. Instead of writing the trace: {e1; e4; e5; e6; e7}, we would for instance write it as:

et:= (tag)*; e1; (tag)*; e4; (tag)*; e5; (tag)*; e6; (tag)*; e7; (tag)*  (Eq. 6)

where the symbol "tag" designates the set of all possible tags that can be added to the execution model. In our case, "tags" corresponds to:

tag:= main_begin + main_end + Fn1_begin + Fn1_end + Fn2_begin + Fn2_end + Fn3_begin + Fn3_end  (Eq. 7)

Once this operation is done, we can then abstract the trace by identifying execution paths that are common to the trace and the found model. This is represented by the following expression:

$t \cap M$  (Eq. 8)

In our example this operation gives the following expression:

main_begin; e1; Fn1_begin; e4; e5; e6; fn1_end; e7; main_end  (Eq. 9)

Events contained between a "_begin" and a "_end" may be abstracted by the corresponding function. Lajeunesse-Robert (2008) shows how to calculate "∩" and "\".

## Case 2 – Only a portion of the trace is available:

### a) Informal description:

So far, we have considered the case where it was possible to follow step by step the execution of the program from the beginning of the program's execution. There will be situations where execution traces will be incomplete; trace elements corresponding to the beginning of the execution will not be available (as for Case 1). This case is more likely to happen than Case 1.

The first thing that must be done in this case is to verify that the incomplete execution trace is included in one or more execution paths of the program's model. Referring to our example (Figure 17), a trace like {e4; e5; e6} would correspond to two execution paths (red lines in Figure 20).

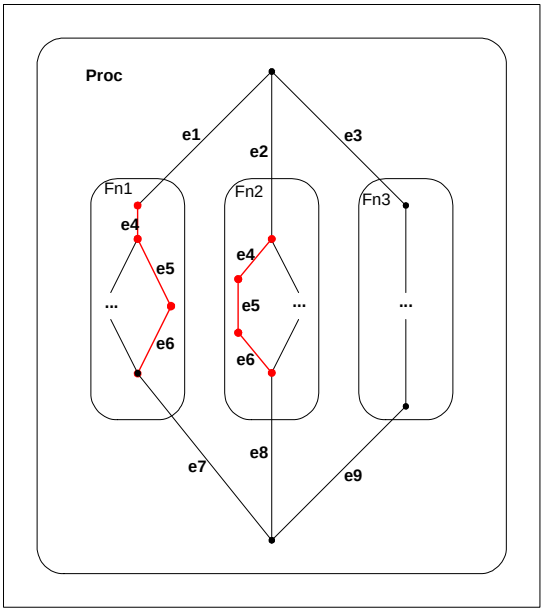


*Figure 20. Example of a program model (D).*

Additional events would then be captured in execution traces over time, making it possible to identify the exact executed path. Once identified, additional trace elements would then confirm this result. Figure 21 shows that e7 or e8 is expected to happen. If the executed function is Fn1, e7 would be observed at the end of Fn1.

The process consists thus in finding (in the execution model) the execution paths to which trace elements correspond, but the process now involves the covering of the whole model (Figure 22). One starts by identifying all possible execution paths that could potentially generate these trace elements. The new execution model is generated from these entry points (not from the main function as in Case 1). The same process as the one presented in Case 1 is then used to carry on the generation of the execution model (based on the next trace element).

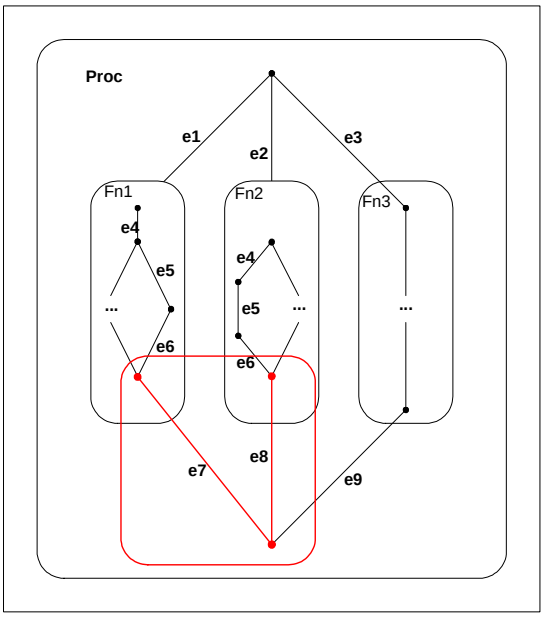


*Figure 21. Example of a program model (E).*

Once the execution path is found, it becomes possible to deduce past trace elements (the ones that were triggered before we started to record elements in the execution trace) based on the full program's execution model. This is illustrated in Figure 22 (red lines). It is then possible to abstract partial execution traces in terms of functions that were called.

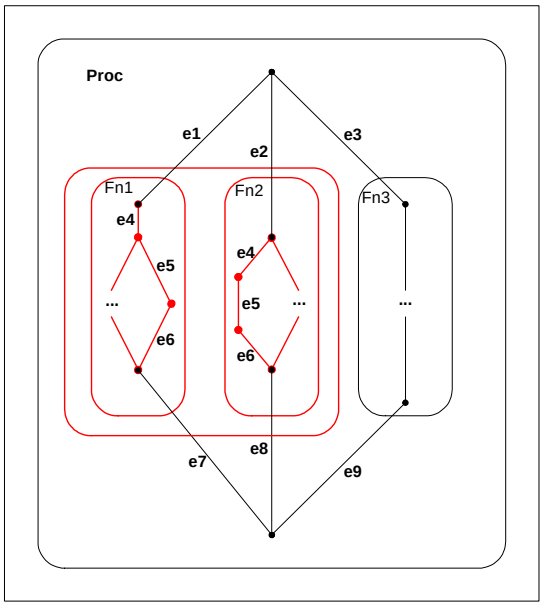


*Figure 22. Example of a program model (F).*

b) **Using the Kleene algebra formalism:**

The Kleene formalism can be used to express a number of theoretical operations that can be done on execution traces as well as on the program's execution model. It provides a means for formal reasoning. The problem of finding if a partial execution trace can be associated with a specific execution path (Case 2) can be solved by searching the whole execution model for these trace elements (or events). In Kleene formalism, this is represented by Eq. 10.

$$\infty; t; \infty \cap M \qquad \text{(Eq. 10)}$$

where "t" represents the incomplete execution trace and "M" the execution model. Actually, the expression "$\infty$ t $\infty$" designates the set of all execution traces that contain the incomplete trace "t". The goal is to find if the set of all traces that can be generated by the model "M" has common traces ("$\cap$") with the set of all traces containing the incomplete trace ("$\infty$; t; $\infty$").

The identification of execution paths in the model (based on trace elements) is done in the same way as described earlier. The model (it is a graph) is searched for trace elements and, once they are found, the corresponding paths (from that location in the graph) are captured. Another way is to identify every execution path that ends with the observed partial execution trace. This is represented by Eq. 11.

$$\infty; t \_| M \qquad \text{(Eq. 11)}$$

where "t" corresponds to the observed incomplete execution trace and "M" the model. The expression "$\infty$ t" designates the set of all traces that end with the incomplete trace "t". To find corresponding execution paths from the model, we identify the set of all traces that begin with this incomplete trace. This is represented by Eq. 12.

$$M \,|\_\, t; \infty \qquad \text{(Eq. 12)}$$

We can identify from the model the execution paths surrounding an incomplete execution trace. This is represented by Eq. 13.

$$(t1 \_| M) \,|\_\, t2 \qquad \text{(Eq. 13)}$$

where "t1" designates the part of the model that we find before the observed incomplete trace, "t2" the part of the model that we find after the observed incomplete trace, and "M" is the model of the program.

**Observations:**

The Kleene algebra is a good theoretical framework for representing (up to a certain limit) programs that are executed on a system and corresponding execution traces. Using this formalism, algebraic expressions can be manipulated and logical reasoning about these expressions can be achieved.

Despite the fact that the Kleene formalism offers means that could be used to solve numerous problems in this domain, it may not be as useful in real operational situations where concrete trace analysis must be effectively and efficiently achieved (in quasi real-time mode). The lack of specificity of this formalism represents a major disadvantage in terms of performance because

algorithms are generic; they were not expressly made for solving specific problems. It appears from this study that dedicated algorithms must be developed.

Nevertheless, the Kleene formalism could be used for certifying that programs have expected behaviours; it is relatively easy to verify that operations on execution traces are valid.

### 3.3.4 Task 4 – Program Analysis Toolkit (PAT) for automating trace analysis

This section describes an academic tool (PAT) that was developed for studying the feasibility of achieving the "automatic abstraction of execution traces" (as described in this chapter).

Technological choices regarding the development of PAT, the description of its functionalities and utilization as well as important observations and future works are presented in this section. More details regarding PAT and the syntax used will be given in a report to be published in 2009.

#### 3.3.4.1 Technical considerations

The technical choices that were made in this task are the following.

- C was the programming language considered.
- The program under study does not contain recursions.
- Only system calls were considered both in the program's execution model and execution traces.

PAT is made up of three main parts: 1) a graphic user interface (GUI), 2) the abstraction part, which achieves trace abstraction from algebraic representations of programs and execution traces, and 3) the model generation part, which generates the program's execution model in the form of an algebraic expression. Following is a brief introduction of each part.

**Part 1: PAT's graphic user interface**

In order to be able to run PAT remotely, PAT's GUI was developed with ASP.NET. Figure 23 and subsequent figures show the PAT GUI.

**Part 2: Abstraction:**

This part of the tool was made with the F# programming language (F#, 2008) for three reasons:

1. F# is a functional programming language. It facilitates the implementation of algorithms that are used to manipulate algebraic expressions.

2. F# is an object-oriented programming language. It allows the structuring of programs in classes and objects, and the use of inheritance.

3. F# is integrated with the Microsoft .NET framework. This framework allows the concurrent use of other programming languages plus quick-to-use tools that are useful in the prototyping context.

It should be mentioned that the current form of PAT does not offer the best performance; F# is an interpreted programming language. In real-world situations, where online analysis must be achieved in quasi real-time, PAT should take the form of an efficient, effective and secure standalone executable program.

This part is divided into three separate but interdependent layers:

- Layer 3: this layer makes use of 1) the lower layers and 2) the notions of program and execution traces to perform trace abstraction operations. In other words, layer 3 uses available algebraic expression operations (layer 2) to perform trace abstraction.
- Layer 2: this layer is used to manipulate algebraic expressions. It contains the definition of all operations that can be performed on algebraic expressions. Operations allow the implementation of many types of program analysis that are based on the algebraic approach.
- Layer 1: this layer offers the higher layers generic useful functionalities.

**Part 3: Model generation from the source code:**

A C programming language parser (ckit, written in SML) is used to generate the model from the source code; the parser transforms the program's source code into an "abstraction syntax tree", which is then used to generate the model. This parser verifies many aspects of the program (syntax, types) and it offers a number of compatibility advantages with the .NET framework.

Another SML module transforms C source code (in the form of an abstraction syntax tree) into an algebraic expression according to the rules defined in Section 3.3.3.

#### 3.3.4.2 PAT's functionalities

PAT's GUI (Figure 23) shows two edit boxes (EB) when it is initially started. These edit boxes allow the user to enter the model of the C program (left EB) or/and the observed execution trace (right EB). Both the model and the trace should take the form of an algebraic expression.

Figure 23 shows a situation where both the program model and the execution trace were entered by the user. PAT starts the analysis of the execution trace according to the program model when the "Process Trace" button is pressed.

Once the analysis is ended, a message is given to the user. This message tells if the execution trace belongs to one of the possible execution paths of the model provided. In this case (in Figure 23, below the "Process Trace" button) the message indicates that the execution trace is part of one of the model execution paths.

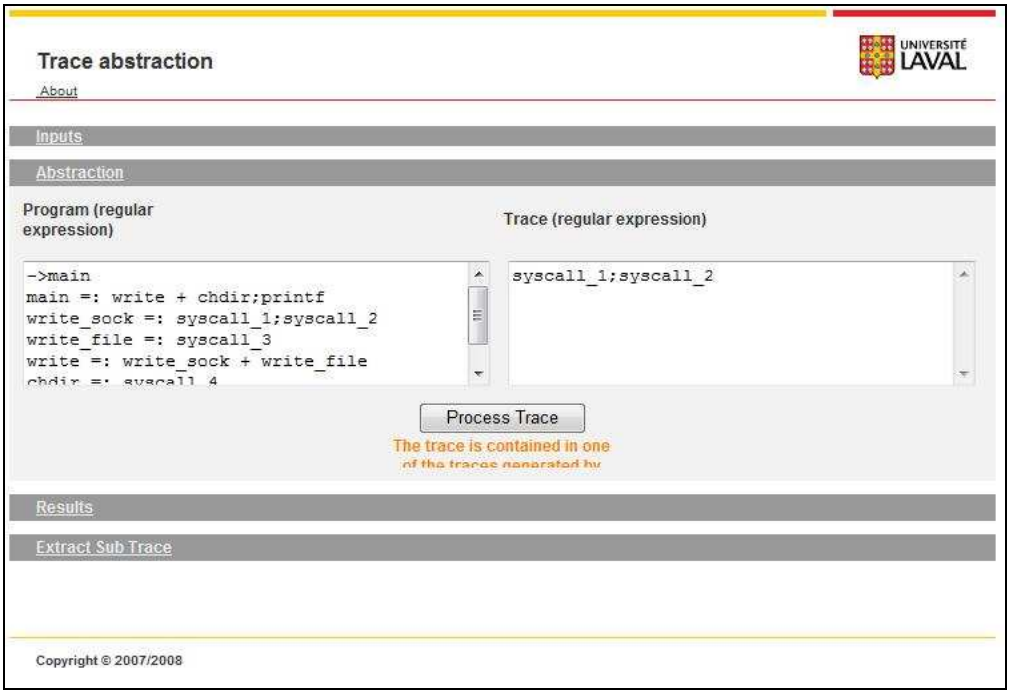


*Figure 23. PAT's graphic user interface.*

In the case where the execution trace is "part of the program model", additional information regarding the analysis process is available by clicking on the "Results" link of the GUI (Figure 24). PAT will then expand the GUI and provide an additional section containing this information. Figure 24 shows the result of this operation with the example provided.

The following information is provided by PAT:

- Abstracted Trace: the trace considered (in abstracted form).
- Sub-process generating traces containing the given trace: the portion of the program that is exactly restricted to the path that generates the trace.
- Sub-process generating traces containing the given trace as sub-trace: the part of the program model that generates the trace.
- Sub-process leading to the trace: the preliminary part of the program model that generates the trace.
- Sub-process following the trace: the ending part of the program model that generates the trace.
- Sub-process not generating the trace: the part of the program model that does not generate the trace.

The information provided in the "Results" section can be useful for many types of trace analysis.


*Figure 24. Trace analysis results.*

PAT also offers the possibility of finding (in the execution path provided) the portion of path that corresponds to a specific function. This functionality is particularly useful when the execution trace is large. The section of the GUI allowing this functionality can be activated by clicking on the "Extract sub Trace" link (Figure 25).

Figure 25 shows an example where the user wanted to identify the part of the execution trace that corresponds to the function "`write_sock`". Corresponding trace elements were generated by clicking on the "Get Sub Trace" button.

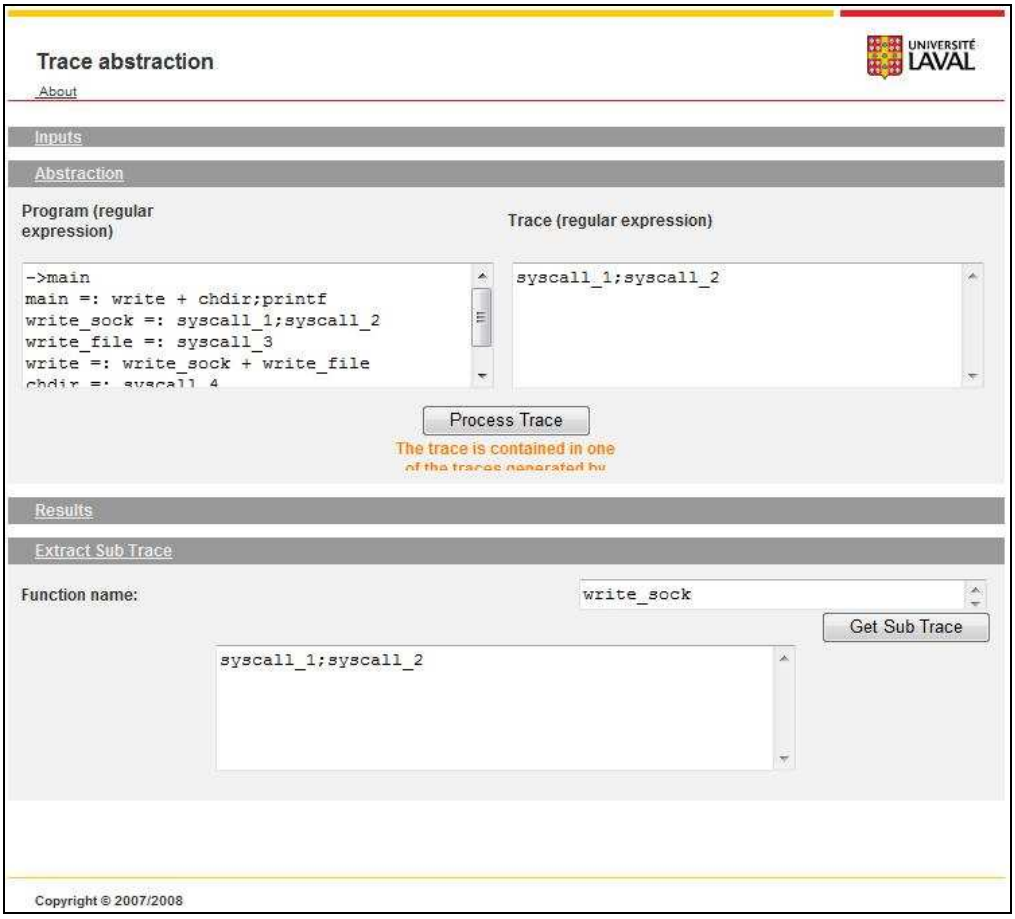


*Figure 25. Identification of functions in execution traces.*

Finally, PAT's conversion module also allows the generation of program models in the form of algebraic expressions. The section of the GUI allowing this transformation can be activated by clicking on the "Inputs" link (Figure 26).

The program C source code is entered (or imported) in the left edit box. The algebraic expression corresponding to the program provided is generated when the "Translate" button is clicked. Figure 26 shows a simple example.

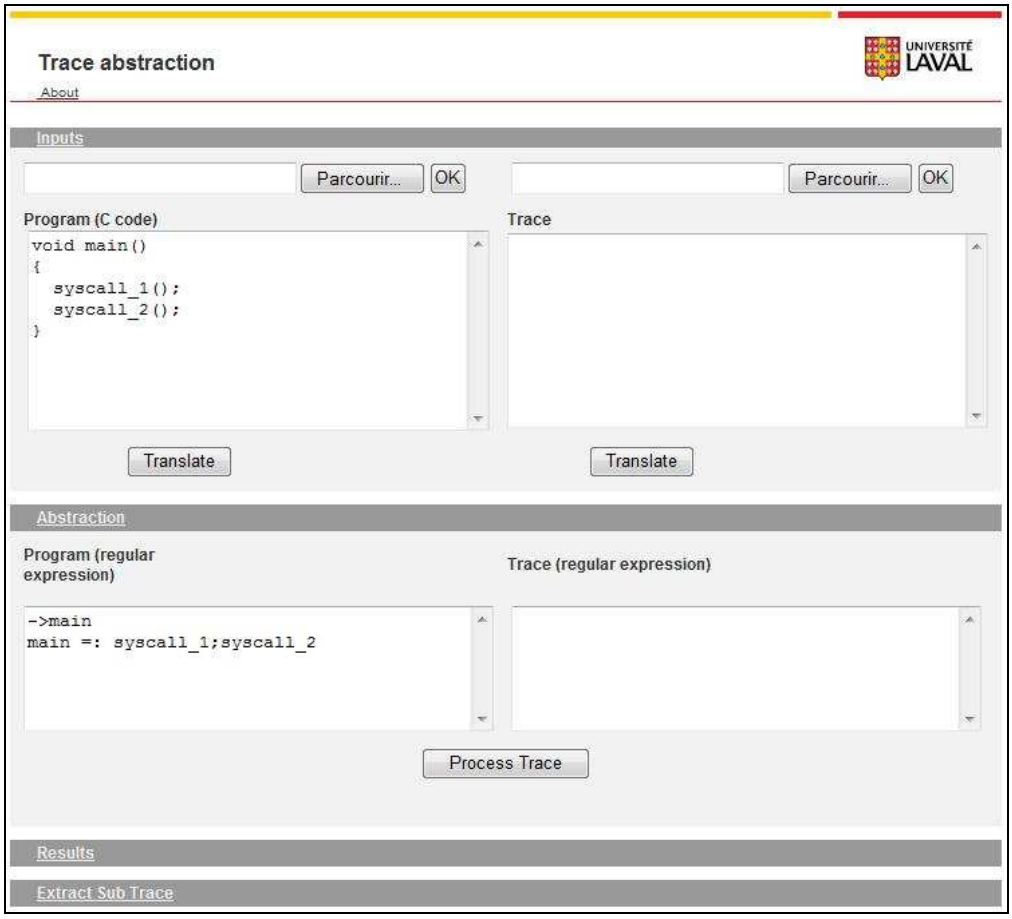


*Figure 26. Translating a C program into an algebraic expression.*

**Observations:**

It was mentioned that PAT consists of three different parts: GUI, Abstraction and Model generation. This prototype could be improved and made autonomous by modifying the current modules and by adding others. Following are some potential improvements.

- Purge all unnecessary behaviours from the program analysis before its model is generated. Unnecessary behaviours are those that are not observed in the execution trace. In our case, only the parts of the program containing calls to system calls would be kept. This preliminary program analysis optimization should be done automatically prior to the model generation process.
- An interface module should be added between the *LTTng* trace generation module and PAT, allowing the appropriate transformation of raw *LTTng* execution traces.
- Preliminary static (and potentially dynamic) program analysis would certainly help improve PAT's functionalities.

# 4 Conclusion and recommendations

Detailed tracing and monitoring of distributed systems is starting to be used in a number of industrial applications. New applications that are executed on heavily used multi-core systems present significant challenges in terms of: 1) efficiently retrieving the monitoring data with minimal disturbance on the system and 2) converting the timestamp of each captured event (originating from different cores) to a common time base. The benefits are however significant; low-disturbance efficient tracing of distributed multi-core systems is the only way to monitor and understand the behaviour and performance (under real load) of these systems. The absence of such a complete and integrated toolset severely limits the speed and efficiency with which such systems can be designed and deployed. This has been identified as a software gap, and it represents a major challenge to those who want to take advantage of the performance and power efficiency gains associated with multi-core systems.

Interestingly, this new technology offers tremendous potential for C2 systems. These systems, when deployed in military or industrial settings, are increasingly multi-core, distributed multi-applications with numerous networked clients and soft real-time expectations. In this context, tracing and offline analysis tools are needed for debugging and tuning. In addition, the low disturbance and detailed level of the data generated by these tools can be used to monitor C2 systems continuously and detect the early signs of several problems, including security attacks. Indeed, low disturbance ensures that monitoring can be permanently active, and minimizes the chances that an attacker will detect the monitoring.

As discussed in the previous two chapters, there were two important technological risks associated with the use of this tracing technology for online monitoring, which was initially targeted for debugging and tuning.

1. First, the architecture of data collection and analysis was optimized for low impact, a posteriori and offline trace retrieval and analysis. The possibility of the generalization of the *LTTng* framework for online tracing was assessed during R&D Thread 1 (Chapter 2).

2. Second, if detailed levels of the captured data are essential for a posteriori analysis, it may represent an overwhelming workload in the context of online analysis. R&D Thread 2 (Chapter 3) addresses this issue by looking at automated algorithms for the online abstraction and analysis of this data.

These two complementary R&D threads were conducted at DRDC Valcartier during summer 2008 and the results were presented at DRDC Valcartier (Lajeunesse-Robert, 2008; Prenoveau, 2008). The aim of the threads was to study and prototype a number of concepts and mechanisms (and to assess their applicability in this new context of online tracing and analysis of C2 systems) before the launch of a major R&D project in this direction (Dagenais 2008b; Couture and Charpentier, 2008).

## 4.1 Main observations

### 4.1.1 R&D Thread 1

In order to extend the tracing and monitoring framework to online data retrieval and display, three elements needed to be developed or adapted.

1. **Support for network transfer**. In the original framework, the tracing data was saved on local files. This capability was extended in this work. The data can now be sent across networks to a remote server. Care had to be taken while flushing data buffers. While buffering is important for performance reasons, periodic flushing of buffers (which fill too slowly) is required to put an upper bound on the latency of tracing data transfer. This was implemented successfully. Some easily solvable issues were encountered: minimizing the number of timers for the periodic flushing and efficiently managing partially filled buffers.

2. **Remote control**. The *LTTng* framework was extended to allow remote control. It is relatively easy to build on currently available secure communication and command execution frameworks like SSH.

3. **Continuous decoding of execution traces**. The feasibility of online display and analysis of tracing data has been assessed as well. The *LTTV* trace viewer already separates the viewing of the online captured raw data from the background computation of the complete trace. A small adjustment is required to continuously display the online data while it is generated. The background computation source code will need to be re-architected in order to allow incremental computation of some specific global properties (e.g. current system state at any point in the trace, average values over some time window for different parameters, etc.).

This R&D thread has successfully identified architectural changes, techniques and algorithms to generalize the *LTTng* framework for continuous and online data gathering and analysis. As a consequence, the risks associated with this critical technical aspect of the R&D project were identified and reduced.

It is recommended that, early in the project, *LTTng* and the associated daemon *lttd* be extended for online tracing. Special care will be required to minimize the number of timers and connections required on multi-core systems. Moreover, the trace collection server will be a critical element in the scalability of this system, and it must be designed accordingly. Early in the R&D project, the *LTTV* viewer must be adapted to allow online viewing of the raw data. The adaptation for online display of the background calculation of higher level synthesized trace information may be done later.

The reading of the tracing information and the computation of higher level information were re-architected in order to: 1) facilitate the integration of tracing data from several sources, and 2) make this data available to a richer integrative framework (Eclipse was chosen for this purpose; Couture et al., 2008). The Eclipse framework will enable the effective exploitation of the captured data through several specialized high-level viewers as well as higher level information on the monitored system (e.g. source code, UML models).

### 4.1.2 R&D Thread 2

In order to achieve effective and efficient analysis of execution traces, three elements needed to be considered.

1. **Simplification of execution traces**. The semantic of trace elements, their inherent logic and the relationship between them have been studied in order to simplify execution traces. For instance, using the information provided by *LTTng* framework it is possible to de-multiplex the events of a raw trace and obtain different sub-traces where each one of them reflects activities related to one specific running process in the system. Moreover, it is possible to find the relationship between processes and the system resources they use (e.g. files), and to capture the different operations that were made on these resources. It is also possible to identify the resources that are shared between processes. However, care must be taken to handle situations where a resource is accessed by different concurrent processes. The analysis of traces coming from concurrent processes is left to future work.

2. **Abstraction of execution traces**. The next step was to study how execution traces can be abstracted into high-level behaviours. In this study, it was assumed that the source codes of both the user's program and the operating system were available. Care must be taken regarding the resolution of execution traces. The resolution must be high enough to allow abstraction.

3. **Implementation of trace abstraction and analysis**. We used the Kleene algebra formalism to represent both the events in execution traces and the model of the executed program. This formalism provides a means for formal reasoning. Trace abstraction has been achieved through manipulations and logical reasoning on algebraic expressions. Two cases have been studied: 1) the case where only a portion of the trace is available, and 2) the case where the trace covers the whole execution of the program. This was implemented successfully as a toolkit (PAT), which can achieve automatic trace analysis, model generation and trace abstraction.

This R&D thread has successfully identified techniques and algorithms to analyze and abstract execution traces. An additional interesting observation is that trace analysis can be improved by using techniques that were developed in other research areas. Intrusion detection systems (IDS) is one example, in which techniques designed to analyze network packets (searching for attack patterns) can be adapted to the context of trace analysis.

As mentioned earlier, trace abstraction requires execution traces that have an appropriate level of resolution. If the level of details is not high enough, some suspicious behaviours can be missed during analysis. If the level is too high, the huge execution trace may become unmanageable (for online analysis). Hence, the resolution of execution traces must be appropriately chosen.

Despite the fact that the Kleene algebra formalism offers an appropriate way to represent abstractions of programs, it remains that in real operational situations, where trace analysis must be effective and efficient, the lack of specificity of this formalism represents a major drawback in terms of performance.

Finally, the successful results of this R&D thread were obtained based on a number of simplifications. Namely, 1) only calls to the kernel interface (system calls) were considered, 2) processes were only studied individually, 3) we assumed that the source codes of both the software and the operating system were available, and 4) we considered only non-recursive functions. These simplifications will have to be reconsidered in future work so that more realistic systems can be handled.

## 4.2 Recommendations for the next R&D efforts

Results and observations from R&D Threads 1 and 2 clearly show the great potential of the proposed R&D project. Each of the components studied (for online monitoring, tracing and analysis of distributed multi-core systems) is clearly feasible and very promising. Four important challenges must be addressed. Actually, they can be considered as recommendations for the next R&D efforts.

1. First, the overhead associated with the *LTTng* framework should be as low as possible to ensure that it is widely applicable and effective. Low overheads do not affect the throughput of the system being monitored, providing a more accurate picture of its behaviour and making it stealthier in the face of sophisticated security attacks, which attempt to detect such defensive detection measures.

2. The second important challenge is to facilitate the integration of the different sources of data (e.g. static and dynamic, kernel and user level trace points) from different processor cores, virtual and physical machines and distributed systems. Moreover, the database structure must allow synthesized information to be added to the raw event traces (e.g. state information, synthesized abstract higher level events, etc.). Some of the problems to be solved are algorithmic (e.g. trace synchronization and conversion to a common time base), while others are architectural or simply involve adaptation between different formats. Nevertheless, integration is important to enable experimentation in the different contexts (e.g. different C2 systems, military or telecom, different operating systems, etc.). The analysis framework must then allow various analysis modules to use and contribute to the shared information database. The analysis modules and underlying algorithms should benefit from a flexible architecture and the availability of detailed and accurate information through a flexible framework.

3. The third important challenge is related to the analysis of execution traces in real operational situations. Traces obtained from concurrent and parallel processes that are executed on distributed multi-core CPUs and which access shared resources must be analyzed. The resolution issue must be addressed as well. As we pointed out in our study, an interesting solution for this problem could be given by using probabilities, which can have a positive impact on the evaluation and approximation of system health.

4. Finally, another important challenge consists in finding the most appropriate formalism for trace analysis, trace abstraction and fault detection. The Kleene algebra formalism is adequate for the certification process but not for the efficient online handling of huge execution traces. A formalism to solve these specific problems must be identified or developed.

This page intentionally left blank.

# References

## A.1 Papers, reports, theses and books


Adam, B. and D. Kozen, 2002. Equational Verification of Cache Blocking in LU Decomposition using Kleene Algebra with Tests. Technical Report TR2002-1865, Computer Science Department, Cornell University, Ithaca, New York, USA, June 2002.

Allegra, A. and D. Kozen, 2001. Kleene Algebra with Tests and Program Schematology. Technical Report 2001-1844, Cornell University, Department of Computer Science, Cornell University, Ithaca, NY, USA, July 2001.

Avizienis, A., B. Randell and C. Landwehr, 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 1.

Benaskeur, A. and D. Blosgett, 2008. Multi-agent coordination techniques for naval tactical combat resources management. DRDC Valcartier TR 2006-784. Defence R&D Canada – Valcartier; July 2008.

Bligh, M., M. Desnoyers and R. Schultz, 2007. Linux kernel debugging on Google-sized clusters. In Proceedings of the 2007 Linux Symposium, Ottawa, Ontario. pp. 29-40.

Bolduc, C., 2006. Oméga Algèbre : Théorie et application en vérification de programmes. Master's Thesis, Université Laval, Quebec, Canada.

Carbone, R., 2006a. Operating system hardware reconfiguration – A case study for Linux. DRDC Valcartier TM 2006-595. Defence R&D Canada – Valcartier; November 2006.

Carbone, R., 2006b. Enterprise Linux licenses – A comparison of licenses between Red Hat and Suse Enterprise Linux. DRDC Valcartier TN 2006-573. Defence R&D Canada – Valcartier; October 2006.

Carbone, R. and R. Charpentier, 2006. Life-Cycle Support for Information Systems Based on Free and Open Source Software. In proceedings of the 11th ICCRTS, paper number I-136. International Command and Control Research and Technology Symposium. ICCRTS, "Coalition Command and Control in the Networked Era", September, 26-28, 2006, Cambridge, UK.

Carbone, R., 2008. Long-term operating system maintenance – A Linux case study. DRDC Valcartier TN 2007-150. Defence R&D Canada – Valcartier; January 2008.

Charpentier, R. and R. Carbone, 2004. Free and Open Source Software – Overview and Preliminary Guidelines for the Government of Canada. DRDC Valcartier ECR 2004-232. Defence R&D Canada – Valcartier; December 2004.

Chen, X., H. Hsieh, F. Balarin, and Y. Watanabe, 2003. Automatic trace analysis for logic of constraints. In Proceedings of the 40th Conference on Design Automation, Anaheim, CA, USA,

June 2-6, 2003. DAC '03. ACM, New York, NY, 460-465. This document could be found in 2008 at: http://doi.acm.org/10.1145/775832.775952

Cohen, E., D. Kozen, and F. Smith, 1996. The complexity of Kleene algebra with tests. Technical Report TR96-1598, Computer Science Department, Cornell University, Ithaca, New York, USA, July 1996.

Cornelissen, B. and L. Moonsen, 2007. Visualizing Similarities in Execution Traces. Proceedings of the 14th Working Conference on Reverse Engineering, October 2007, Vancouver, BC, Canada.

Couture, Mathieu, 2005. Détection d'intrusions et analyse passive de réseaux. Master's Thesis, Université Laval, Quebec, Canada.

Couture, M., 2007. Complexity and chaos – State-of-the-art – Overview of theoretical concepts. DRDC Valcartier TM 2006-453. Defence R&D Canada – Valcartier; August 2007.

Couture, M. and R. Charpentier, 2008. Multi-Core Monitoring and Soft Redundancy for Cyber-Attack Protection. DRDC Valcartier project proposal and presentation. TAG 15B, Ottawa, October 14, 2008, DRDC Valcartier TM 2006-453. Defence R&D Canada – Valcartier; October 2008.

Couture, M., M. Dagenais, D. Toupin, R. Charpentier, G. Matni, M. Desnoyers, P.M. Fournier, 2008. Monitoring and tracing of critical software systems – State of the work and project definition. DRDC Valcartier TM 2008-144. Defence R&D Canada – Valcartier; June 2008.

Corbet, J., A. Rubini, G. Kroah-Hartman, 2005. *Linux Device Drivers*. O'Reilly Media, 3rd edition, ISBN: 0-696-00590-3, 615 pages.

Dagenais, M., 2008a. State-of-the-Art – Tracing and Monitoring of Multi-core Systems. École Polytechnique de Montréal. This document could be found in 2008 at: http://ltt.polymtl.ca/tracingwiki/index.php/Main_Page

Dagenais, M., 2008b. Tracing and Monitoring Tools for Distributed Multi-Core Systems. NSERC Application for Grant, Form 101. Area of application: 1207/801. Natural Sciences and Engineering Research Council of Canada. 42 pages.

Desnoyers, M. and M. Dagenais, 2006a. Low disturbance embedded system tracing with Linux Trace Toolkit next generation. In Proceedings of the 2006 Consumer Electronics Linux Forum, San Jose, California, USA.

Desnoyers, M. and M. Dagenais, 2006b. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In Proceedings of the 2006 Linux Symposium, Ottawa, Ontario, Canada, pp. 209-224.

Desnoyers, M. and M. Dagenais, 2008. LTTng: Tracing across execution layers, from the Hypervisor to user-space. In Proceedings of the 2008 Linux Symposium, Ottawa, Ontario, Canada, pp. 101-106.

Desharnais, J., B. Möller, and G. Struth, 2004. Modal Kleene Algebra and Applications – A Survey. Journal on Relational Methods in Computer Science (JoRMiCS), 1:93–131, 2004.

Desharnais, J., B. Möller, and G. Struth, 2006. Kleene algebra with domain. ACM Transactions on Computational Logic, 7(4):798–833. ACM, New York, NY, USA, 2006.

Dexter, K., 1990. On Kleene algebras and closed semirings. In Rovan, editor, Proceedings of Mathematical Foundations of Computer Science, Volume 452 of Lecture Notes in Computer Science, pages 26–47, Banska-Bystrica, Slovakia. Springer-Verlag.

Dexter, K., 1994. A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events. Information and Computation, 110:366–390, May 1994.

Dexter, K., 1997a. Kleene Algebra with Tests. ACM Transactions on Programming Languages and Systems, 19(3):427–443, May 1997.

Dexter, K., 1997b. On the Complexity of Reasoning in Kleene Algebra. In Logic in Computer Science (LICS'97), pages 152–162, 1997.

Dexter, K., 1998. Typed Kleene algebra. Technical Report TR98-1669, Computer Science Department, Cornell University, Ithaca, NY, USA, March 1998.

Dexter, K. and M.-C. Patron, 2000. Certification of compiler optimizations using Kleene algebra with tests. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luis Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, Proceedings of the 1st International Conference on Computational Logic (CL2000), Volume 1861 of Lecture Notes in Artificial Intelligence, pages 568–582, London, UK, July 2000. Springer-Verlag.

Dexter, K., 2003. Kleene Algebras with Tests and the Static Analysis of Programs. Technical Report 2003-1915, Computer Science Department, Cornell University, Ithaca, NY, USA, November 2003.

Dexter, K., 2004a. Introduction to Kleene Algebra. Computer Science Department, Cornell University, Ithaca, NY, USA.

Dexter, K., 2004b. Introduction to Kleene Algebra. Lecture 19 (see Dexter, 2004), Computer Science Department, Cornell University, Ithaca, NY, USA, Spring 2004.

Dexter, K., 2004. Introduction to Kleene Algebra . Lecture 2 (see Dexter, 2004), Cornell University, Ithaca, NY, USA, Spring 2004.

DRDC, 2006. Defence S&T Strategy. Science and Technology for a Secure Canada. ISBN: D2-186/2006 978-0-662-49705-9, NDID: A-JS-007-000/AF-004. This document could be found in 2008 at: http://descartes.drdc-rddc.gc.ca/default.aspx

Ehm, T., 2004. Pointer Kleene Algebra. In Springer Berlin / Heidelberg, editor, Tools and Algorithms for the Construction and Analysis of Systems, Volume 3051/2004 of Lecture Notes in Computer Science, pages 99:111, May 2004.

Ellison, B. and C. Woody, 2007a. Scale: System Development Challenges. US Department of Homeland Security, Build Security In – Setting a higher standard for software assurance. This document could be found in 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/system-strategies/882-BSI.html

Ellison, B. and C. Woody, 2007b. Introduction to System Strategies. US Department of Homeland Security, Build Security In – Setting a higher standard for software assurance. This document could be found in 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/system-strategies/883-BSI.html

Ellison, B. and R. Creel, 2007. Acquisition Overview: The Challenges. US Department of Homeland Security, Build Security In – Setting a higher standard for software assurance. This document could be found in 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/acquisition/893-BSI.html

Ezust, S. A. and G. v. Bochmann, 1995. An automatic trace analysis tool generator for Estelle specifications. SIGCOMM Computer Communication Review, 25, 4 (Oct. 1995), 175-184. This document could be found in 2008 at: http://doi.acm.org/10.1145/217391.217428

Fischer, M., J. Oberleitner, H. Gall and T. Gschwind, 2005. System Evolution Tracking through Execution Trace Analysis. Proceedings of the 13th International Workshop on Program Comprehension, St. Louis, MO, USA.

Fusco, J., 2007. The Linux Programmer's Toolbox. Prentice Hall, ISBN 0132198576

Hamou-Lhadj, A., 2005. Techniques to Simplify the Analysis of Execution Traces for Program Comprehension. Ph. D. thesis. Ottawa-Carleton Institute for Computer Science, School of Information Technology and Engineering, University of Ottawa, 171 pages. This document could be found in 2008 at: http://users.encs.concordia.ca/~abdelw/HamouLhadjPhDThesis.pdf

Hardin, C. and D. Kozen, 2002. On the Elimination of Hypotheses in Kleene Algebra with Tests. Technical Report 2002-1879, Department of Computer Science, Cornell University, Ithaca, NY, USA, November 2002.

Hardin, C. and D. Kozen, 2003. On the complexity of the Horn theory of REL. Technical Report TR2003-1896, Computer Science Department, Cornell University, Ithaca, NY, USA, May 2003.

Hardin, C., 2005. Proof Theory for Kleene Algebra. In LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05), pages 290–299, Washington, DC, USA, 2005. IEEE Computer Society.

Heikkila, E. and J. Gulliksen, 2007. Multi-Core Computing in Embedded Applications. VDC Market Research. September 2007.

Jackson, D., M. Thomas, and L.I. Millett, 2007. Software for Dependable Systems: Sufficient Evidence? Committee on Certifiably Dependable Software Systems, National Research Council. Daniel Jackson, Martyn Thomas, and Lynette I. Millett, Editors. ISBN: 0-309-66738-0, 148 pages. This document could be found in 2008 at: http://www.nap.edu/catalog/11923.html.

Jipsen, P., 2004. From Semirings to Residuated Kleene Lattices. Studia Logica, 76(2):291–303, 2004.

Kamal, A.-H. and D. Kozen, 2007. Local Variable Scoping and Kleene Algebra with Tests. Journal of Logic and Algebraic Programming, 2007. DOI : 10.1016/j.jlap.2007.10.007.

Kot, L. and D. Kozen, 2005. Kleene Algebra and Bytecode Verification. In Fausto Spoto, Editor, Proceedings of the 1st Workshop on Bytecode Semantics, Verification, Analysis, and Transformation (Bytecode'05), Edinburgh, Scotland, pages 201–215, April 2005.

Ktari, B., F. Lajeunesse-Robert, and C. Bolduc, 2008. Solving Linear Equations in *-continuous Action Lattices. In Bernhard Möller, editor, Relations and Kleene Algebra in Computer Science, volume 4988 of Lecture Notes in Computer Science, pages 289–303. Springer, April 2008.

Lajeunesse-Robert, F., 2008. Résolution d'équations en algebra de Kleene – Applications à l'analyse de programme. Master's Thesis, Université Laval, Quebec, Canada.

Langevine, L., 2002. Automated analysis of CLP (FD) program execution traces. Proceedings of the International Conference on Logic Programming. Lecture Notes in Computer Science. Springer-Verlag. Pages 470-471.

Lau, T., P. Domingos and D.S. Weld, (2003). Learning Programs from Traces using Version Space Algebra. Proceedings of the 2nd International Conference on Knowledge Capture. ACM, New York, NY, USA.

Leiß, H., 2006. Kleene Modules and Linear Languages. Journal of Logic and Algebraic Programming, 66:185–194, 2006.

Lianjiang, F., 2005. Exploration and Visualization of Large Execution Traces. M. Sc. thesis. Ottawa-Carleton Institute for Computer Science, University of Ottawa, 127 pages. This document could be found in 2008 at: http://www.site.uottawa.ca/~tcl/gradtheses/lfu/LfuThesisJun30-2005Final.pdf

Linger, R.C. and A. P. Moore, 2001. Foundations for Survivable System development: Service Traces, Intrusion Traces, and Evaluation Models. Carnegie Mellon University Software Engineering Institute, Survivability Systems. Technical Report CMU/SEI-2001-TR-029, ESC-TR-2001-029. Carnegie Mellon University, Pittsburgh, PA, USA.

Mathieu, V., 2006. Vérification des systèmes à pile au moyen des algèbres de Kleene. Master's Thesis, Université Laval, Quebec, Canada, 2006.

Neumann, P.G., 2000. Practical Architectures for Survivable Systems and Networks. Phase-Two Final Report based upon work supported by the U.S. Army Research Laboratory (ARL), under contract DAKF11-97-C-0020, SRI International, 209 pages.

Neumann, P.G., 2004. Principled Assuredly Trustworthy Composable Architectures. Final report. Contract number N66001-01-C-8040. DARPA Order No. M132. SRI Project P11459. 222 pages.

Paxson, V., 1997. Automated Packet Trace Analysis of TCP Implementations. ACM SIGCOMM Computer Communication Review. Volume 27 Issue 4, ACM.

Schneider, Fred B., 1999. Trust in Cyberspace. Committee on Information Systems Trustworthiness, Commission on Physical Sciences, Mathematics, and Applications, National Research Council. Fred B. Schneider, Editor. ISBN: 0-309-51970-5, 352 pages. This document could be found in 2008 at: http://www.nap.edu/catalog/6161.html.

Yaghmour, K. and M.R. Dagenais, 2000. Measuring and characterizing system behavior using kernel-level event logging. In Proceedings of the USENIX 2000 Annual Technical Conference, San Diego, California, USA, pp. 13-26.

Yaghmour, K., 2001. Analyse de performance et caractérisation de comportement à l'aide d'enregistrement d'événements noyau. Master's Thesis. École Polytechnique de Montréal, Montreal, Quebec.

Zanussi, T., K. Yaghmour, R. Wisniewski, R. Moore, and M. Dagenais, 2003. Relayfs: An efficient unified approach for transmitting data from kernel to user space. In Proceedings of the Ottawa Linux Symposium, Ottawa, Ontario, pp. 519-531.

## A.2 Presentations

Lajeunesse-Robert, F., 2008. Abstraction de traces d'exécution de programmes. Science and Technology Matinées. DRDC Valcartier, August 19, 2008.

Prenoveau, F., 2008. Traçage de systèmes d'information en ligne. Science and Technology Matinées. DRDC Valcartier, August 19, 2008.

## A.3 Web sites

Cisco_IDS, 2008. http://www.cisco.com/warp/public/cc/pd/sqsw/sqidsz/index.shtml

F#, 2008. http://research.microsoft.com/fsharp/fsharp.aspx

Foldoc, 2008: http://foldoc.org/

Kernel, 2008: http://kernel.org/

LTTng, 2008: http://ltt.polymtl.ca/

Micro_httpd, 2008. http://www.acme.com/software/micro_httpd/

Relayfs, 2008: http://relayfs.sourceforge.net/

WikipediaWS, 2008. http://en.wikipedia.org

# Annex B Tracing for distributed multi-core systems – Motivations and drivers

A tutorial/workshop on monitoring and tracing of information systems[37] was held in January 2008 in Montréal (Couture et al., 2008)[38]. Several of the most advanced players[39] in the areas of advanced communications, information management and computer security discussed the state of the art with regard to current tools, associated problems and unmet needs. The main goals of this tutorial/workshop were:

- *to ask these international experts in these domains to describe the leading edge of knowledge relative to tracing and monitoring of distributed computer systems (SMP or single/multi-core), and*
- *to ask advanced users to present the most challenging problems they are facing, and potential solutions.*

The most promising avenues for solutions were identified with the guidance of key industrial, governmental and academic researchers in the field. Key findings of the tutorial/workshop are presented in this section.

## B.1 Initial drivers of the effort

The long-term vision key points that were initially proposed to the participants of the workshop were the following:

- *low-overhead instrumentation is critical to most real-world applications;*
- *equally important are in-lab debug and in-field (on-line) monitoring;*
- *many applications are distributed (i.e. multi CPUs) and threaded in multi-core;*

*The technologies to be promoted are:*

- *Linux shall be considered as the most appropriate OS for this work[40];*
- *standard protocols, formats and integration frameworks shall also be considered; and*

*Open-Source reference implementations shall be encouraged as well.*

[37] This event corresponds to step four of the used methodology (Section 1.2).
[38] The text in *italic* in this section was literally reproduced verbatim from Couture et al. (2008).
[39] Representatives from Defence R&D Canada at Valcartier, Enea, Ericsson, Freescale, IBM, MontaVista, Nokia, Rational, Red Hat, Oracle, Wind River and Zealcore were present.
[40] A wide variety of Operating Systems are encountered but Linux appears to be the most appropriate OS for R&D demonstration since it is shared by a large community of users and expected to increase in importance in the future.

## B.2 Identified long term challenges

The following long-term challenges were identified by the participants during the workshop:

1. **Full spectrum trace visualization**: Full spectrum trace visualization appears to be desirable in most analyses (i.e. from silicon, hypervisor, OS, VM, and simulator up to user application). Abstraction towards a modelling level was also expressed as a valuable visualization enabler in some system designs and analyses.

2. ***Multiple CPUs and multi-core within an integrated CPU****: Understanding interactions between multiple CPUs and multi-core within an integrated CPU is also mandatory to complex system analysis and in particular to ensure scalability of performance.*

3. ***Reusability and comparability of functions and traces****: Tracing and monitoring functions should be designed in ways that enable regressive testing and periodic (repetitive) maintenance.*

4. ***Measures of system health and interventions****: Exploitation of low-level instrumentation to assess general system health of on-line components (at high level) and the activation of reactive measures when appropriate are also perceived as being critical to autonomous complex systems. Ultimately, a "continuous process of feedback-directed adaptation and optimization" could emerge from such capabilities.*

5. ***Forensics****: Conditioning captured data for forensics exploitation could also be highly appreciated for criminal investigations when malfunctions are suspected to originate from an attack on a key component of a critical infrastructure.*

6. ***Ensure technology adoption in a broader community of users by prioritizing "ease-of-use".***

## B.3 Identified Areas of R&D

Six key areas of R&D were identified during the tutorial/workshop. They are listed in the following lines (this text comes from Dagenais, 2008b).

***1) Adaptative fault probing:***

*Static probe sites may be inserted at compilation time and remain dormant until activated at runtime, typically to generate tracing information upon need. Dynamic probes can be added at runtime, to adapt the system behavior, for example to trace various parts of the operating system for diagnosing a problem. DTrace [8] is probably the best known recent implementation of static probe sites and dynamic probes. SystemTap [9], currently under development, offers a similar functionality for dynamic probes under Linux. The most important characteristics of static probe sites and dynamic probes is their ability to be inserted anywhere (including in interrupt and even non-maskable interrupt context), their low overhead (minimal performance hit when dormant and when activated) and their low disturbance (do not change the real-time behavior).*

*The group under the supervision of Michel Dagenais has started working on static probe sites, providing an initial implementation for the mainline Linux kernel, named Kernel Markers, used by thousands of developpers and installed on millions of computers around the world. The challenge is to simultaneously minimize the number of execution cycles required for a probe, including the effect of the added instructions on the memory cache, while not interfering with the real-time response of the system, and enabling the activation of probes even when the program may be simultaneously accessed by several processors on a multi-core system. To achieve this, atomic operations local to a CPU, per CPU buffers and data structures, and special techniques for code patching online multi-threaded binary executable code may be used.*

*Some interesting preliminary results have been obtained with the Kernel Markers and LTTng [2]. The interoperability with SystemTap dynamic probes has also been planned. Further research and development to refine and optimize the underlying algorithms will be needed to provide a robust, low disturbance and extremely efficient infrastructure for adaptative fault probing within the operating system kernel and for user level applications (D1.1 to D1.4). While this infrastructure will be prototyped for Linux, at operating system and user level, the same algorithms will be adapted for a number of other operating systems (e.g. BSD) in use at Ericsson, Defence R&D Canada and elsewhere to trace heterogeneous distributed systems (D1.5 and D1.6).*

***2) Multi-level, multi-core distributed traces synchronization:***

*The probes installed at the different software layers (hypervisor, operating system, virtual machine, system libraries, applications) may be used to provide monitoring and tracing data. Each processor, with its own local clock, then generates a steady flow of events. These events, from multiple cores on each system, and from several distributed systems, must then be collected and stored efficiently. The events coming from the different cores must be synchronized and allow navigation through the possibly huge traces. The currently available trace visualization tools have often targeted detailed traces for small real-time embedded systems, or much less detailed system logs for larger systems. As a result, none of the systems evaluated for the 2008 Tracing Summit [4] were capable of handling traces of more than a few tens of megabytes. The Linux Trace Toolkit Viewer, developed at Ecole Polytechnique, is capable of handling huge traces of several gigabytes or more. However, a new architecture is required to handle huge traces while allowing the collection of traces from multiple systems and embedded devices (R1.2, R1.4 and R1.5), for both online and a posteriori offine analysis and viewing.*

*More importantly, further work is required to develop algorithms for the synchronization of events coming from multiple nodes, multiple cores and even multiple virtual machines (R1.1 and R1.3). Existing tracing tools for distributed systems often use coarse level events, for which local clocks differences may not be a problem. Tools for tracing newer distributed real-time systems [13] rely on the local clocks synchronization and incur a significant loss of accuracy. A posteriori synchronization of traces allows more accurate drift estimation because the network delay variations are amortized over a large number of message exchanges and a long period of time [14].*

***3) Trace abstraction, analysis and correlation:***

*The detailed event lists gathered independently from several processors need to be processed to extract higher level information suitable for analysis at higher levels of abstraction. This may be achieved by detecting self similar sections of the trace, or sections matching use case maps, which may correspond to higher level utility operations. Different metrics may be used (frequency of occurence, number of different contexts in which a sequence is used) in order to estimate the likelihood of a sequence to represent a higher level abstract utility operation. These new techniques have been used successfully for higher level nested method calls traces [15]. The context is significantly different in system level traces where asynchronous work queues and interrupts mix with simpler nested calls.*

*As a first step, the possible events abstraction will be examined on a single system (H1.1 to H1.5). For instance, a sequential file reading operation may be the abstract representation of numerous, possibly out of order, disk block reads. New efficient algorithms will be developed based on pattern detection techniques to abstract out the content of low-level traces generated from multicores systems. Pattern detection algorithms rely on matching criteria to assess the extent to which various parts of a trace can be deemed similar. A key activity of this milestone is to study how existing matching criteria can be applied to low-level traces and if there is a need to create new ones. The resulting algorithms will vary in the way existing (or new) matching criteria are combined and weighed.*

*This trace reduction and abstraction step is useful in itself to reduce the size of traces and simplify their understanding for the human viewer. It also allows the automated comparison of multiples traces, to quantify and classify the divergences, establishing equivalence in result even if implementation differs. This will be useful in different contexts. It may be used for regression analysis between traces sampled periodically, monitoring the system performance and detecting any degradation, or to insure that subsequent revisions of a software system have not introduced programming errors. It may also be used to compare the traces of redundant servers, performing the same work but through distinct software implementations, in order to detect any malfunction possibly caused by a security breach.*

*As a second step, the abstraction algorithms will be enhanced and extended to analyze multilevel distributed systems (H2.1 to H2.5). For example, client-server requests may be identified by finding specific sequences of message send and receive events (e.g. DNS UDP packets). This is different from the distributed traces synchronization research thread where simple pairs of matching lower level packet send and receive events (typically TCP) are identified wherever possible to obtain common time reference points.*

*A significant challenge is to cope with the variability and uncertainty associated with the approximate time synchronization of events from independently clocked cores and computers, and the unobvious matching of request and response pairs, particularly in the presence of non connection oriented protocols with lost packets and retransmissions.*

***4) Automated fault identification:***

*By carefully examining execution traces of an information system, experts can detect problematic behaviors that are related to software design defects, inefficiencies as well as malicious activities. Examples of such faulty behaviors may include: excessive swapping, lock contention, undue*

*latency, inefficient task scheduling, attempts to erase system logs, modification of system files, etc.*

*Mechanisms allowing fault detection already exist in Intrusion Detection Systems (IDS). Among others, they analyze network packet traces and look for attack patterns. Many of these use specialized languages to represent faulty conditions, scenarios, patterns, etc. These languages have different flavors, some are: domain specific (Panoptis, Snort, NeVO), imperative languages (ASAX, BRO), finite states (STAT, IDIOT, BSML), expert systems (P-BEST, LAMBDA), temporal logic (LogWeaver, Monid, Chronicles).*

*A similar approach is chosen in this project to provide an online flexible automated fault identification mechanism for execution traces. The main goal is to allow systems to trigger alarms during operations when specified problematic conditions, scenarios or patterns are detected in execution traces. Such detection system will significantly improve the decision making process as well as thoughtful analysis and response, while maximizing time for risk mitigation.*

*Problematic conditions that are found in execution traces will first be studied in depth (K1.1, K2.1 and R2.1) in order to clearly identify what semantic must be represented by the language. A state of the art will then be done to identify a number of complementary languages and potential solutions that could be used (K1.2). New algorithms, patterns and techniques will then be developed to detect a wide range of non timing critical problems (K1.3 to K1.5), then for timing critical problems (K2.2 to K2.5), including denial of service cyber-attacks, and finally for multi-level distributed systems specific problems (R2.2 to R2.5).*

***5) System health monitoring and corrective measure activation:***

*System irregularities and overall performance degradation [16] can be observed and measured to a new level of awareness by the enabling technologies described above (trace abstraction, analysis and correlation, automated fault identification, adaptive fault probing, etc). The objective of this research and development thread is to define and to validate quantitative measures that may be used to assess global system health and appropriate activation of corrective measures.*

*System health can be evaluated using an array of different, often complementary, approaches. A more traditional approach is using the low level (trace) metrics [17] or statistics computed directly in the probes. Similar metrics may be built on higher level abstracted events. The comparison of correlated traces from redundant systems will use different techniques to extract the differences, and measure their size and significance; different techniques such as measuring the edit distance, used to study the temporal evolution of source code in a project, or to detect cloned source code blocks, may be used.*

*A model of healthy behavior may be described or deduced from system characterization through the analysis of several traces. Thereafter, any indication that the trace of a monitored system diverges from the healthy model is then flagged as suspicious. Additionally, strict access rules for the different system resources by the different processes may be defined and checked during the trace monitoring; these access rules may be much more fine grained and thus precise than what is supported by the operating system. The pattern languages developed for automated fault*

*identification and traces abstraction may be used to characterize important aspects of the system health.*

*The resulting system health or reliability measurement can be highly informative to the system administrator to increase the level of surveillance or to activate additional protective and reactive measures such as logging more information, modifying packet filtering rules, inserting and/or activating probes dynamically, modifying system behaviour or simply shutting down some computers in order to protect them from hacking or malicious exploitation (D3.1 to D3.8).*

***6) Trace directed modeling:***

*Many benefits can be achieved by processing trace events so as to reconstruct a higher-level model, such as state machine or an interaction model. Necessary operations to build such a model include filtering less-relevant items (e.g. utilities) from the trace; the process also builds on trace abstraction, analysis and correlation. Key challenges are to manage the scope and to determine the appropriate abstractions; this can be guided by the presence of an original UML model used to create the system (if available). An example of the above is taking a series of transmissions in a system and recognizing that it represents a single transaction (L1.1 and L1.5).*

*The reconstructed model can then applied in several ways: 1) It can be used for anomaly detection and analysis by comparing it to either the original UML model, or to models generated from previous correct or anomalous runs. 2) Delays, timing and resource usage can be analysed at the level of the abstract entities in the model. 3) Various visualization techniques can be applied to the model to allow engineers to obtain insights. 4) It becomes possible to change the model (e.g. change the amount or distribution of resources, or change a condition) and estimate the effect on the system's performance or behaviour (L1.6 to L1.8).*

*Building on the synchronization of distributed multi-level multi-core traces, the basic operating system model currently available in the LTTng viewer [6] will be extended to represent multi-level (system and user level virtual machines) distributed systems by modeling explicitly the relationships and interactions between these real and virtual systems (D2.1 to D2.5). These interactions will then be analyzed to identify and report critical paths detailing the delay components, for instance explaining the time elapsed between a query and a response on a distributed system.*

# Annex C Used tracing softwate – Overview

An overview of *LTTng* is presented in this annex. The reader should keep in mind that this description refers to the 2006 version of the software. Since then, the software has been the object of intensive evolution phases designed to make it part of the mainline of the Linux kernel. The reader will find a more complete description in Desnoyers and Dagenais (2008).

## C.1 Overview of the *LTTng* architecture

The first version of *LTTng* was actually called *LTT* (Yagmour, 2001). The main goal of this work was to build a set of tools that could provide their users an exact continually updated picture of the dynamical aspects of their system. The knowledge of performance issues and associated causes were at the centre of this R&D effort. At its beginning in 2001, execution of the first version *LTT* was already showing low relative CPU costs of 2.5%.

The arrival of the new version of *LTT*, "*LTT* new generation" (*LTTng*), confirmed the necessity of keeping these CPU costs as low as possible when dealing with increasingly complex systems. *A low impact, high performance tracing system* appears to be *the only tool capable of collecting the information produced by instrumenting the whole system, while not changing significantly the studied system behaviour and performance* (Desnoyers and Dagenais, 2006b).

One of the challenges that were encountered at the beginning of the *LTTng* development was to obtain a *flexible and extensible tracer with precise timestamps, across multiple architectures, running from several MHx to several GHz, some being multi-processors*. Its flexibility would facilitate the addition of new instrumentation points as well as plugins for the analysis and display of the traced data.

The architecture of *LTTng* can be described using five interlinked architectural views; 1) Control; 2) Data flow; 3) Instrumentation; 4) Event type registration; and 5) Tracing. These views are introduced in the following lines. This text (and Figure 27) essentially was borrowed from Desnoyers and Dagenais (2006b); the reader is invited to consult this paper (and subsequent) for more details.

**1) Control:**

There are three main parts in the *LTTng* architecture (Figure 27):

- ***lttctl***: *a user space command-line application that is used to control tracing. It starts lttd and controls kernel tracing behavior through a librarymodule bridge which uses a netlink socket*.
- ***lttd***: *a user space daemon that waits for trace data and writes it to disk.*
- **a kernel part that controls kernel tracing**.

*The core module of LTTng is* ***ltt-core****. This module is responsible for a number of LTT control events.* It controls the following helper modules:

- ***ltt-heartbeat***: this modeule *generates periodic events in order to detect and account for cycle counters overflows, thus allowing a single monotonically increasing time base even if shorter 32-bit (instead of 64-bit) cycle counts are stored in each event.*
- ***ltt-facilities***: this modeule *lists the facilities (collection of event types) currently loaded at trace start time.*
- ***ltt-statedump***: this modeule *generates events to describe the kernel state at trace start time (processes, files…).*

*A builtin kernel object, **ltt-base**, contains the symbols and data structures required by builtin instrumentation. This includes principally the tracing control structures.*

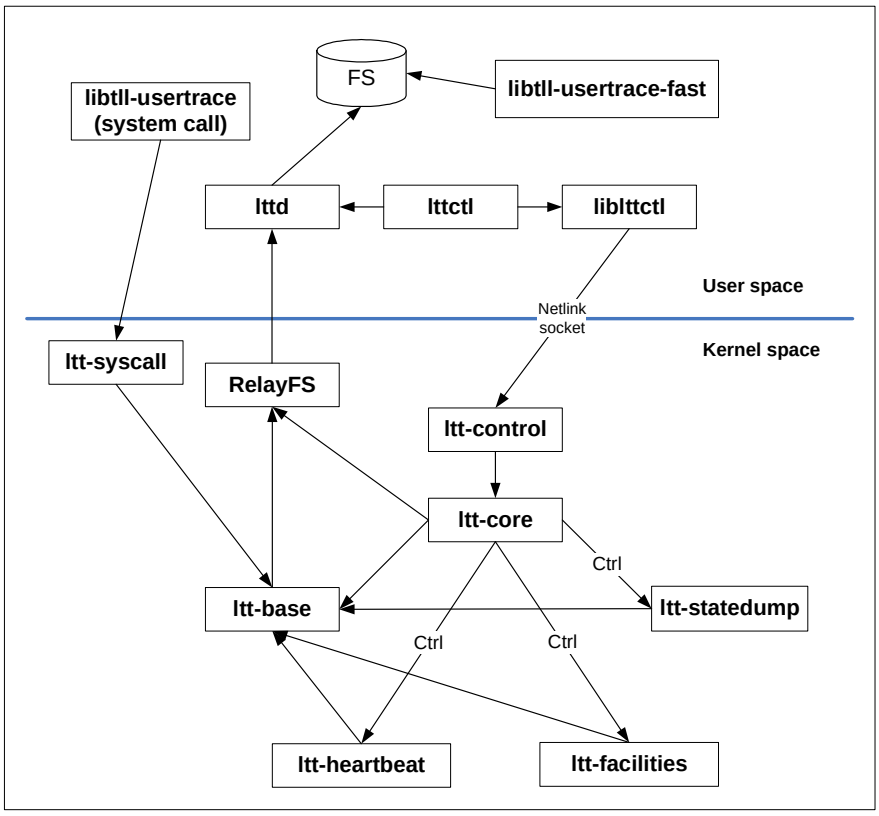


*Figure 27. Architecture of the tracing software LTTng.*

**2) Data flow:**

The data flows from the kernel to the user space in the following manner. *All data is written through **ltt-base** into **RelayFS** circular buffers* (Zanussi et al., 2003; RelayFS, 2008). *When subbuffers are full* (please refer to Section 2.4.1 of this document for definition of *buffers*, *subbuffers* and *channels*), *they are delivered to the **lttd** disk writer daemon. **lttd** is a standalone multithreaded daemon which waits on **RelayFS** channels (files) for data by using the poll file operation. When it is awakened, it locks the channels for reading by using a relay buffer get ioctl.*

*At that point, it has exclusive access to the subbuffer it has reserved and can safely write it to disk. It should then issue a relay buffer put ioctl to release it so it can be reused.*

*A side-path,* ***libltt-usertrace-fast****, running completely in user space, has been developed for high throughput user space applications which need high performance tracing.* (…) *Both* ***lttd*** *and the* ***libltt-usertrace-fast*** *companion process currently support disk output, but should eventually be extended to other media like network communication.* (Chapter **Error! Reference source not found.** addresses partially this problematic).

**3) Instrumentation:**

*LTTng instrumentation* (…) *consists in an XML event description that is used both for automatically generating tracing headers and as data metainformation in the trace files. These tracing headers implement the functions that must be called at instrumentation sites*[41] *to log information in traces. Most common types are supported in the XML description: fixed size integers, host size integers (int, long, pointer, size_t), floating point numbers, enumerations, and strings. All of these can be either host or network byte ordered. It also supports nested arrays, sequences, structures, and unions. The tracing functions, generated in the tracing headers, serialize the C types given as arguments into the LTT trace format. This format supports both packed and aligned data types. A record generated by a probe hit is called an event. Event types are grouped in facilities. A facility is a dynamically loadable object, either a kernel module for kernel instrumentation or a user space library for user space instrumentation. An object that calls instrumentation should be linked with its associated facility object.*

**4) Event type registration:**

*Event type registration is centralized in the* ***lttfacilities*** *kernel object* (…)*. It controls the rights to register specific type of information in traces. For instance, it does not allow a user space process using the ltt-usertrace API to register facilities with names conflicting with kernel facilities. The* ***ltt-heartbeat*** *built-in object and the* ***lttstatedump*** *also have their own instrumentation to log events. Therefore, they also register to* ***ltt-facilities****, just like standard kernel instrumentation. Registered facility names, checksums and type sizes are locally stored in* ***ltt-facilities*** *so they can be dumped in a special low traffic channel at trace start. Dynamic registration of new facilities, while tracing is active, is also supported. Facilities contain information concerning the type sizes in the compilation environment of the associated instrumentation. For instance, a facility for a 32-bit process would differ from the same facility compiled with a 64-bit process from its long and pointer sizes.*

**5) Tracing:**

(…) *traced information must have its metainformation registered into* ***ltt-facilities****.* (…) *The tracing path has the biggest impact on system behavior because it is called for every event. Each event recorded uses* ***ltt-base****, container of the active traces, to get the pointers to* ***RelayFS*** *buffers.* Used lockless mechanisns make them reentrant, scalable, and precise.

[41] In this paper, **call site**, is defined as *the original code from the instrumented program where the tracing function is called* and **instrumentation site** is defined as *the tracing function itself*.

## C.2 Installation of *LTTng*

The procedure for installing the tracing software *LTTng* is listed in this section. It was made especially for the Ubuntu Desktop 8.10 LTS (June, 2008).

1. ##########################################################################

2. **Tools that are needed for this installation process**

3. ##########################################################################

4. **Create a working directory and change directory:**

    a. mkdir ~/lttng_stuff/

    b. cd ~/lttng_stuff

5. **Download, from the LTTng web site (LTTng, 2008), the following three files: (if the last version of these files are choosen, the reader should make sure the version numbers correspond to the one of the Linux kernel that will be used).**

    a. patch-2.6.25.4-lttng-0.10-pre55.tar.bz2

    b. ltt-control-0.48-27022008.tar.gz

    c. lttv-0.10.0-pre11-10032008.tar.gz

6. **Download from Kernel (2008) the Linux kernel. The reader should make sure that the version number of the kernel is the same as the one of the three files that were downloaded at step 5. In our example, the following file should be downloaded:**

    a. linux-2.6.25.4.tar.bz2

7. **Install development tools that will be needed. The following line triggers this installation:**

    a. sudo apt-get install build-essential quilt libncurses-dev fakeroot kernel-package libgtk2.0-dev libpopt-dev

8. ##########################################################################

9. **Compile and install the downloaded Linus kernel**

10. ##########################################################################

11. **Extract file from downloaded files using the following commands:**

    a. tar xjf linux-2.6.25.4.tar.bz2

    b. tar xjf patch-2.6.25.4-lttng-0.10-pre55.tar.bz2

12. **Apply the patches with the following commands:**

    a. cd linux-2.6.25.4

    b. ln -s ../patch-2.6.25.4-lttng-0.10-pre55 patches

    c. quilt push -a

13. **Configure the Linux kernel. Here, we copy the default configuration.**

    a. cp /boot/config-`uname -r` .config

    b. make menuconfig

**14. The last command (#13-b) will laungh a menu allowing the selection of options. The reader should make sure that the following options are selected:**

a. General Setup --->

i. [X] Activate markers

b. Linux Trace Toolkit --->

i. [X] LTTng fine-grained timestamping

ii. [X] Linux Trace Toolkit Instrumentation Support

ii. <*> Linux Trace Toolkit Relay+DebugFS Support

iii. <*> Linux Trace Toolkit Serializer

iv. <*> Linux Trace Toolkit Marker Control

v. <*> Linux Trace Toolkit Tracer

vi. [*] Align Linux Trace Toolkit Traces

vii. [*] Support trace extraction from crash dump

viii. [ ] Write hearbeat event to shrink trace (EXPERIMENTAL)

ix. <M> Linux Trace Toolkit Netlink Controller

x. <M> Linux Trace Toolkit State Dump

**15. Following options should be disabled in order to lower the size of the kernel and its modules. It is important thay the reader make sure that these elements be desactivated.**

c. Kernel Hacking --->

i. [ ] Enable unused/obsolete exported symbols

ii. [ ] Kernel debugging

iii. [ ] Compile kernel with debug info

**16. Once this configuration process has ended, quit and save. The kernel may now be compiled using the following commands:**

a. make-kpkg clean

b. fakeroot make-kpkg --initrd --append-to-version=-lttng-0.10-pre55 kernel_image

**17. The kernel compilation process may take a long time. When it is finished, a package containing the kernel and its modules will be created and saved in the directory containing the source of the kernel. Use the following commands to install the new compiled kernel.**

a. cd ..

b. sudo dpkg -i linux-image-2.6.25.4-lttng-0.10-pre55_2.6.25.4.deb

**18. The computer must be restarted in order to load the new kernel. While booting, the use will have to select among two or many compiled kernels. S/he should select the LTTng enabled kernel.**

**19.** ##########

**20. Configuration of the file system “debugfs”**

21. ##############################################################################

22. **Create a directory in /mnt that will allow the mounting of the debugfs file system.**

    a. sudo mkdir /mnt/debugfs

23. **Edit the file "/etc/fstab" and add the following line at the end of the file, save and quit the editor:**

    a. debugfs /mnt/debugfs debugfs rw 0 0

24. **Mount the file system debugfs using the following command:**

    a. sudo mount /mnt/debugfs

25. ##############################################################################

26. **Compilation and installation of ltt-control**

27. **(ltt-control is the application allowing the control of the tracing).**

28. ##############################################################################

29. **Compile and install ltt-control with the following commands:**

    a. tar xzf ltt-control-0.48-27022008.tar.gz

    b. cd ltt-control-0.48-27022008

    c. ./configure

    d. make

    e. sudo make install

30. ##############################################################################

31. **Compilation and installation of lttv**

32. **(lttv is the visualisation tool).**

33. ##############################################################################

34. **Compile and install lttv with the use of the following commands:**

    a. tar xzf lttv-0.10.0-pre11-10032008.tar.gz

    b. cd lttv-0.10.0-pre11-10032008

    c. ./configure

    d. make

    e. sudo make install

35. **Finally, update the dynamical library index with the following command:**

    a. sudo ldconfig

## C.3 How to use

1. ##############################################################################

2. Mettre un mot de passe root

3. ##############################################################################

4. In order to control the tracing with lttv, the password of lttv should be configured with tha one of the root. Use the following command:

   a. sudo passwd

   b. Enter the password twice

5. ##############################################################################

6. Testing LTTng

7. ##############################################################################

8. Load modules and arm markers with the following commands (these commands must be retyped after every start of the computer):

   a. sudo modprobe ltt-control

   b. sudo modprobe ltt-statedump

   c. sudo ltt-armall

**9. LTTng is now ready to be used. The graphical interface of LTTng (LTTV) may be lauched with the following command:**

   a. lttv-gui

**10. To start a trace, click on the red/yellow/green icon that is located in the tool box. En ter the root password and click on start.**

11. To end tracing, click on stop. The application should then ask for loading the trace into its viewer.

## C.4 List of *LTTng* active markers (or probes)

The following list corresponds to the one that can be found in the last version of *LTTng* (LTTng, 2008).

1. marker: kernel_arch_kthread_create format: "pid %ld fn %p"

2. marker: kernel_arch_trap_exit format: " "

3. marker: kernel_arch_trap_entry format: "trap_id %d ip #p%ld"

4. marker: statedump_idt_table format: "irq %d address %p symbol %s"

5. marker: kernel_arch_ipc_call format: "call %u first %d"

6. marker: kernel_arch_syscall_exit format: "ret %ld"

7. marker: kernel_arch_syscall_entry format: "syscall_id %d ip #p%ld"

8. marker: kernel_irq_entry format: "irq_id %u kernel_mode %u ip %lu"

9. marker: kernel_irq_exit format: "handled #1u%u"

10. marker: kernel_softirq_entry format: "softirq_id %lu"

11. marker: kernel_softirq_exit format: "softirq_id %lu"

12. marker: kernel_softirq_raise format: "softirq_id %u"

13. marker: kernel_tasklet_low_entry format: "func %p data %lu"

14. marker: kernel_tasklet_low_exit format: "func %p data %lu"

15. marker: kernel_tasklet_high_entry format: "func %p data %lu"

16. marker: kernel_tasklet_high_exit format: "func %p data %lu"

17. marker: kernel_kthread_stop format: "pid %d"

18. marker: kernel_kthread_stop_ret format: "ret %d"

19. marker: kernel_sched_wait_task format: "pid %d state %ld"

20. marker: kernel_sched_try_wakeup format: "pid %d state %ld"

21. marker: kernel_sched_wakeup_new_task format: "pid %d state %ld"

22. marker: kernel_sched_schedule format: "prev_pid %d next_pid %d prev_state %ld"

23. marker: kernel_sched_migrate_task format: "pid %d state %ld dest_cpu %d"

24. marker: kernel_send_signal format: "pid %d signal %d"

25. marker: kernel_process_free format: "pid %d"

26. marker: kernel_process_exit format: "pid %d"

27. marker: kernel_process_wait format: "pid %d"

28. marker: kernel_process_fork format: "parent_pid %d child_pid %d child_tgid %d"

29. marker: kernel_timer_itimer_expired format: "pid %d"

30. marker: kernel_timer_itimer_set format: "which %d interval_sec %ld interval_usec %ld value_sec %ld value_usec %ld"

31. marker: kernel_timer_set format: "expires %lu function %p data %lu"

32. marker: kernel_timer_update_time format: "jiffies #8u%llu xtime_sec %ld xtime_nsec %ld walltomonotonic_sec %ld walltomonotonic_nsec %ld"

33. marker: kernel_timer_timeout format: "pid %d"

34. marker: kernel_printk format: "ip %lu"

35. marker: kernel_vprintk format: "loglevel %c string %s ip %lu"

36. marker: kernel_module_free format: "name %s"

37. marker: kernel_module_load format: "name %s"

38. marker: mm_wait_on_page_start format: "pfn %lu bit_nr %d"

```
39. marker: mm_wait_on_page_end format: "pfn %lu bit_nr %d"
40. marker: mm_huge_page_free format: "pfn %lu"
41. marker: mm_huge_page_alloc format: "pfn %lu"
42. marker: mm_handle_fault_entry format: "address %lu ip #p%ld write_access %d"
43. marker: mm_handle_fault_exit format: "res %d"
44. marker: mm_page_free format: "order %u pfn %lu"
45. marker: mm_page_alloc format: "order %u pfn %lu"
46. marker: mm_swap_in format: "pfn %lu filp %p offset %lu"
47. marker: mm_swap_out format: "pfn %lu filp %p offset %lu"
48. marker: mm_swap_file_close format: "filp %p"
49. marker: mm_swap_file_open format: "filp %p filename %s"
50. marker: fs_buffer_wait_start format: "bh %p"
51. marker: fs_buffer_wait_end format: "bh %p"
52. marker: fs_exec format: "filename %s"
53. marker: fs_ioctl format: "fd %u cmd %u arg %lu"
54. marker: fs_open format: "fd %d filename %s"
55. marker: fs_close format: "fd %u"
56. marker: fs_lseek format: "fd %u offset %ld origin %u"
57. marker: fs_llseek format: "fd %u offset %lld origin %u"
58. marker: fs_read format: "fd %u count %zu"
59. marker: fs_write format: "fd %u count %zu"
60. marker: fs_pread64 format: "fd %u count %zu pos %llu"
61. marker: fs_pwrite64 format: "fd %u count %zu pos %llu"
62. marker: fs_readv format: "fd %lu vlen %lu"
63. marker: fs_writev format: "fd %lu vlen %lu"
64. marker: fs_select format: "fd %d timeout #8d%lld"
65. marker: fs_pollfd format: "fd %d"
66. marker: ipc_msg_create format: "id %ld flags %d"
67. marker: ipc_sem_create format: "id %ld flags %d"
```

```
68. marker: ipc_shm_create format: "id %ld flags %d"

69. marker: input_event format: "type %u code %u value %d"

70. marker: net_dev_xmit format: "skb %p protocol #2u%hu"

71. marker: net_dev_receive format: "skb %p protocol #2u%hu"

72. marker: net_insert_ifa_ipv4 format: "label %s address #4u%u"

73. marker: net_del_ifa_ipv4 format: "label %s address #4u%u"

74. marker: net_insert_ifa_ipv6 format: "label %s a15 #1x%c a14 #1x%c a13 #1x%c a12 #1x%c
    a11 #1x%c a10 #1x%c a9 #1x%c a8 #1x%c a7 #1x%c a6 #1x%c a5 #1x%c a4 #1x%c a3 #1x%c a2
    #1x%c a1 #1x%c a0 #1x%c"

75. marker: net_insert_ifa_ipv6 format: "label %s a15 #1x%c a14 #1x%c a13 #1x%c a12 #1x%c
    a11 #1x%c a10 #1x%c a9 #1x%c a8 #1x%c a7 #1x%c a6 #1x%c a5 #1x%c a4 #1x%c a3 #1x%c a2
    #1x%c a1 #1x%c a0 #1x%c"

76. marker: net_socket_sendmsg format: "sock %p family %d type %d protocol %d size %zu"

77. marker: net_socket_recvmsg format: "sock %p family %d type %d protocol %d size %zu"

78. marker: net_socket_create format: "sock %p family %d type %d protocol %d fd %d"

79. marker: net_socket_call format: "call %d a0 %lu"
```

# Annex D The *micro_httpd* application

Following pages contain a listing of the *micro_httpd* application.

## D.1 Source code of the *micro_httpd* application

The source code of the *micro_httpd* application is listed “as is” in this section.

```
1.  /* micro_httpd - really small HTTP server
2.  ** Copyright © 1999,2005 by Jef Poskanzer <jef@mail.acme.com>.
3.  ** All rights reserved.
4.  ** Redistribution and use in source and binary forms, with or without
5.  ** modification, are permitted provided that the following conditions
6.  ** are met:
7.  ** 1. Redistributions of source code must retain the above copyright
8.  **    notice, this list of conditions and the following disclaimer.
9.  ** 2. Redistributions in binary form must reproduce the above copyright
10. **    notice, this list of conditions and the following disclaimer in the
11. **    documentation and/or other materials provided with the distribution.
12. ** THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
13. ** ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
14. ** IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
15. ** ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
16. ** FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
17. ** DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
18. ** OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
```

```
19. ** HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
20. ** LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
21. ** OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
22. ** SUCH DAMAGE.
23. */
24. #include <sys/types.h>
25. #include <unistd.h>
26. #include <stdlib.h>
27. #include <stdio.h>
28. #include <string.h>
29. #include <dirent.h>
30. #include <ctype.h>
31. #include <time.h>
32. #include <sys/stat.h>
33. #define SERVER_NAME "micro_httpd"
34. #define SERVER_URL "http://www.acme.com/software/micro_httpd/"
35. #define PROTOCOL "HTTP/1.0"
36. #define RFC1123FMT "%a, %d %b %Y %H:%M:%S GMT"
37. /* Forwards. */
38. static void file_details( char* dir, char* name );
39. static void send_error( int status, char* title, char* extra_header, char* text );
40. static void send_headers( int status, char* title, char* extra_header, char* mime_type, off_t length, time_t mod );
41. static char* get_mime_type( char* name );
```

```
42. static void strdecode( char* to, char* from );
43. static int hexit( char c );
44. static void strencode( char* to, size_t tosize, const char* from );
45. //*****************************************************
46. int main( int argc, char** argv )
47. {
48.
49.     char line[10000], method[10000], path[10000], protocol[10000], idx[20000], location[20000], command[20000];
50.     char* file;
51.     size_t len;
52.     int ich;
53.     struct stat sb;
54.     FILE* fp;
55.     struct dirent **dl;
56.     int i, n;
57.     if ( argc != 2 )
58.             send_error( 500, "Internal Error", (char*) 0, "Config error - no dir specified." );
59.     if ( chdir( argv[1] ) < 0 )
60.             send_error( 500, "Internal Error", (char*) 0, "Config error - couldn't chdir()." );
61.     if ( fgets( line, sizeof(line), stdin ) == (char*) 0 )
62.             send_error( 400, "Bad Request", (char*) 0, "No request found." );
63.     if ( sscanf( line, "%[^ ] %[^ ] %[^ ]", method, path, protocol ) != 3 )
64.             send_error( 400, "Bad Request", (char*) 0, "Can't parse request." );
```

```
65.     while ( fgets( line, sizeof(line), stdin ) != (char*) 0 ){
66.             if ( strcmp( line, "\n" ) == 0 || strcmp( line, "\r\n" ) == 0 )
67.             break;
68.     }
69.     if ( strcasecmp( method, "get" ) != 0 )
70.             send_error( 501, "Not Implemented", (char*) 0, "That method is not implemented." );
71.     if ( path[0] != '/' )
72.             send_error( 400, "Bad Request", (char*) 0, "Bad filename." );
73.     file = &(path[1]);
74.     strdecode( file, file );
75.     if ( file[0] == '\0' )
76.             file = "./";
77.     len = strlen( file );
78.     if ( file[0] == '/' || strcmp( file, ".." ) == 0 || strncmp( file, "../", 3 ) == 0 || strstr( file, "/../" ) != (char*)
    0 || strcmp( &(file[len-3]), "/.." ) == 0 )
79.             send_error( 400, "Bad Request", (char*) 0, "Illegal filename." );
80.     if ( stat( file, &sb ) < 0 )
81.             send_error( 404, "Not Found", (char*) 0, "File not found." );
82.     if ( S_ISDIR( sb.st_mode ) {
83.             if ( file[len-1] != '/' ){
84.                     (void) snprintf(location, sizeof(location), "Location: %s/", path );
85.                     send_error( 302, "Found", location, "Directories must end with a slash." );
86.             }
87.             (void) snprintf( idx, sizeof(idx), "%sindex.html", file );
```

```
88.             if ( stat( idx, &sb ) >= 0 ){
89.                     file = idx;
90.                     goto do_file;
91.             }
92.             send_headers( 200, "Ok", (char*) 0, "text/html", -1, sb.st_mtime );
93.             (void)  printf(  "<html><head><title>Index  of  %s</title></head>\n<body  bgcolor=\"#99cc99\"><h4>Index  of
    %s</h4>\n<pre>\n", file, file );
94.             n = scandir( file, &dl, NULL, alphasort );
95.             if ( n < 0 )
96.                     perror( "scandir" );
97.             else
98.                     for ( i = 0; i < n; ++i )
99.                             file_details( file, dl[i]->d_name );
100.                    (void)  printf(  "</pre>\n<hr>\n<address><a  href=\"%s\">%s</a></address>\n</body></html>\n",  SERVER_URL,
    SERVER_NAME );
101.            }
102.            else{
103.                    do_file:
104.                    fp = fopen( file, "r" );
105.                    if ( fp == (FILE*) 0 )
106.                            send_error( 403, "Forbidden", (char*) 0, "File is protected." );
107.                    send_headers( 200, "Ok", (char*) 0, get_mime_type( file ), sb.st_size, sb.st_mtime );
108.                    while ( ( ich = getc( fp ) ) != EOF )
109.                            putchar( ich );
```

```
110.            }
111.            (void) fflush( stdout );
112.            exit( 0 );
113.    }
114.    //*********************************************************
115.    static void file_details( char* dir, char* name )
116.    {
117.            static char encoded_name[1000];
118.            static char path[2000];
119.            struct stat sb;
120.            char timestr[16];
121.            strencode( encoded_name, sizeof(encoded_name), name );
122.            (void) snprintf( path, sizeof(path), "%s/%s", dir, name );
123.            if ( lstat( path, &sb ) < 0 )
124.                    (void) printf( "<a href=\"%s\">%-32.32s</a>    ???\n", encoded_name, name );
125.            else {
126.                    (void) strftime( timestr, sizeof(timestr), "%d%b%Y %H:%M", localtime( &sb.st_mtime ) );
127.                    (void) printf( "<a href=\"%s\">%-32.32s</a>     %15s %14lld\n", encoded_name, name, timestr, (int64_t)
    sb.st_size );
128.            }
129.    }
130.    //*********************************************************
131.    static void send_error( int status, char* title, char* extra_header, char* text )
132.    {
```

```
133.            send_headers( status, title, extra_header, "text/html", -1, -1 );
134.            (void) printf(  "<html><head><title>%d  %s</title></head>\n<body  bgcolor=\"#cc9999\"><h4>%d  %s</h4>\n",  status,
    title, status, title );
135.            (void) printf( "%s\n", text );
136.            (void) printf( "<hr>\n<address><a href=\"%s\">%s</a></address>\n</body></html>\n", SERVER_URL, SERVER_NAME );
137.            (void) fflush( stdout );
138.            exit( 1 );
139.    }
140.    //*******************************************************
141.    static void send_headers( int status, char* title, char* extra_header, char* mime_type, off_t length, time_t mod )
142.    {
143.            time_t now;
144.            char timebuf[100];
145.            (void) printf( "%s %d %s\015\012", PROTOCOL, status, title );
146.            (void) printf( "Server: %s\015\012", SERVER_NAME );
147.            now = time( (time_t*) 0 );
148.            (void) strftime( timebuf, sizeof(timebuf), RFC1123FMT, gmtime( &now ) );
149.            (void) printf( "Date: %s\015\012", timebuf );
150.            if ( extra_header != (char*) 0 )
151.                    (void) printf( "%s\015\012", extra_header );
152.            if ( mime_type != (char*) 0 )
153.                    (void) printf( "Content-Type: %s\015\012", mime_type );
154.            if ( length >= 0 )
```

```
155.                    (void) printf( "Content-Length: %lld\015\012", (int64_t) length );
156.            if ( mod != (time_t) -1 ){
157.                    (void) strftime( timebuf, sizeof(timebuf), RFC1123FMT, gmtime( &mod ) );
158.                    (void) printf( "Last-Modified: %s\015\012", timebuf );
159.            }
160.            (void) printf( "Connection: close\015\012" );
161.            (void) printf( "\015\012" );
162.    }
163.    //*******************************************************
164.    static char* get_mime_type( char* name )
165.    {
166.            char* dot;
167.            dot = strrchr( name, '.' );
168.            if ( dot == (char*) 0 )
169.                    return "text/plain; charset=iso-8859-1";
170.            if ( strcmp( dot, ".html" ) == 0 || strcmp( dot, ".htm" ) == 0 )
171.                    return "text/html; charset=iso-8859-1";
172.            if ( strcmp( dot, ".jpg" ) == 0 || strcmp( dot, ".jpeg" ) == 0 )
173.                    return "image/jpeg";
174.
175.            if ( strcmp( dot, ".gif" ) == 0 )
176.                    return "image/gif";
177.            if ( strcmp( dot, ".png" ) == 0 )
```

```
178.                return "image/png";
179.          if ( strcmp( dot, ".css" ) == 0 )
180.                return "text/css";
181.          if ( strcmp( dot, ".au" ) == 0 )
182.                return "audio/basic";
183.          if ( strcmp( dot, ".wav" ) == 0 )
184.                return "audio/wav";
185.          if ( strcmp( dot, ".avi" ) == 0 )
186.                return "video/x-msvideo";
187.          if ( strcmp( dot, ".mov" ) == 0 || strcmp( dot, ".qt" ) == 0 )
188.                return "video/quicktime";
189.          if ( strcmp( dot, ".mpeg" ) == 0 || strcmp( dot, ".mpe" ) == 0 )
190.                return "video/mpeg";
191.          if ( strcmp( dot, ".vrml" ) == 0 || strcmp( dot, ".wrl" ) == 0 )
192.                return "model/vrml";
193.
194.          if ( strcmp( dot, ".midi" ) == 0 || strcmp( dot, ".mid" ) == 0 )
195.                return "audio/midi";
196.          if ( strcmp( dot, ".mp3" ) == 0 )
197.                return "audio/mpeg";
198.          if ( strcmp( dot, ".ogg" ) == 0 )
199.                return "application/ogg";
200.          if ( strcmp( dot, ".pac" ) == 0 )
```

```
201.                    return "application/x-ns-proxy-autoconfig";
202.            return "text/plain; charset=iso-8859-1";
203.    }
204.    //*******************************************************
205.    static void strdecode( char* to, char* from )
206.    {
207.            for ( ; *from != '\0'; ++to, ++from ) {
208.                    if ( from[0] == '%' && isxdigit( from[1] ) && isxdigit( from[2] ) ) {
209.                            *to = hexit( from[1] ) * 16 + hexit( from[2] );
210.                            from += 2;
211.                    }
212.                    else
213.                            *to = *from;
214.            }
215.            *to = '\0';
216.    }
217.    //*******************************************************
218.    static int hexit( char c )
219.    {
220.            if ( c >= '0' && c <= '9' )
221.                    return c - '0';
222.            if ( c >= 'a' && c <= 'f' )
223.                    return c - 'a' + 10;
```

```
224.          if ( c >= 'A' && c <= 'F' )
225.                 return c - 'A' + 10;
226.          return 0;             /* shouldn't happen, we're guarded by isxdigit() */
227.   }
228.   //**************************************************
229.   static void strencode( char* to, size_t tosize, const char* from )
230.   {
231.          int tolen;
232.          for ( tolen = 0; *from != '\0' && tolen + 4 < tosize; ++from ) {
233.                 if ( isalnum(*from) || strchr( "/_.-~", *from ) != (char*) 0 ) {
234.                        *to = *from;
235.                        ++to;
236.                        ++tolen;
237.                 }
238.                 else
239.                 {
240.                        (void) sprintf( to, "%%%02x", (int) *from & 0xff );
241.                        to += 3;
242.                        tolen += 3;
243.                 }
244.          }
245.          *to = '\0';
246.   }
```

## D.2 First level abstraction of the application *micro_httpd*

```
int main( ... )
{
	if ( ... )
		send_error( ... );
	chdir( ... );
	if ( ... )
		send_error( ... );
	fgets( ... );
	if ( ... )
		send_error( ... );
	if ( ... )
		send_error( ... );
	fgets( ... );
	while ( ... )
		fgets( ... );
	if ( ... )
		send_error( ... );
	if ( ... )
		send_error( ... );
	if ( ... )
		send_error( ... );
	stat( ... );
	if ( ... )
		send_error( ... );
	if ( ... )
	{
		if ( ... )
			send_error( ... );
		stat( ... )
		if ( ... )
			goto do_file;
		send_headers( ... );
		scandir( ... );
		if ( ... )
			perror( ... );
		else
		for ( ... )
			file_details( ... );
	}
	else
	{
		do_file:
```

```
                fopen( ... );
                if ( ... )
                        send_error( ... );
                send_headers( ... );
                while ( ... )
                        putchar( ... );
        }
        fflush( ... );
        exit( ... );

}
```

# Annex E Example of an execution trace

An example of an execution trace involving the running of the *micro_httpd* application in the user space of the Linux kernel is shown below. Comments (“#”) were added directly to the listing to help the reader follow the suite of events. The reader may refer to Annex D for the listing of the *micro_httpd* application.

It should be noted that this example is intended only to give an idea of the content of a trace. Other execution traces involving different focuses and resolutions can be produced by activating/de-activating *LTTng* probes in the system. All examples used throughout this document originate from this listing.

```
1.  ###############################################################################

2.  # http request from “micro_httpd” server (mounted on xinets): "GET /index.html".

3.  ###############################################################################

4.  list_process_state: 5270.943605493 (/tmp/trace-httpd/control/processes_0), 4974, 0, , , 0, 0x0, MODE_UNKNOWN { pid = 4566,
    parent_pid = 1, name = "xinetd", type = 0, mode = 5, submode = 0, status = 5, tgid = 4566 }

5.  #

6.  # File descriptors for xinetd

7.  # Note that stdin, stdout and stderr point toward '/dev/null' (xinetd is a deamon)

8.  # socket:[9854] (fd=7) is possibly the listening socket (port 80)

9.  list_file_descriptor: 5270.945015874 (/tmp/trace-httpd/cpu_0), 4974, 4974, lttctl, , 4973, 0x0, MODE_UNKNOWN { filename =
    "/dev/null", pid = 4566, fd = 0 }

10. list_file_descriptor: 5270.945017342 (/tmp/trace-httpd/cpu_0), 4974, 4974, lttctl, , 4973, 0x0, MODE_UNKNOWN { filename =
    "/dev/null", pid = 4566, fd = 1 }

11. list_file_descriptor: 5270.945018858 (/tmp/trace-httpd/cpu_0), 4974, 4974, lttctl, , 4973, 0x0, MODE_UNKNOWN { filename =
    "/dev/null", pid = 4566, fd = 2 }

12. list_file_descriptor: 5270.945021038 (/tmp/trace-httpd/cpu_0), 4974, 4974, lttctl, , 4973, 0x0, MODE_UNKNOWN { filename =
    "pipe:[9851]", pid = 4566, fd = 3 }

13. list_file_descriptor: 5270.945022718 (/tmp/trace-httpd/cpu_0), 4974, 4974, lttctl, , 4973, 0x0, MODE_UNKNOWN { filename =
    "pipe:[9851]", pid = 4566, fd = 4 }
```

```
14. list_file_descriptor: 5270.945024729 (/tmp/trace-httpd/cpu_0), 4974, 4974, lttctl, , 4973, 0x0, MODE_UNKNOWN { filename =
    "socket:[9860]", pid = 4566, fd = 5 }

15. list_file_descriptor: 5270.945026921 (/tmp/trace-httpd/cpu_0), 4974, 4974, lttctl, , 4973, 0x0, MODE_UNKNOWN { filename =
    "socket:[9854]", pid = 4566, fd = 7 }

16. ###############################################################################

17. # Beginning of the execution trace.

18. # xinet was waiting, it just received a connexion on port 80 (HTTP)

19. ###############################################################################

20. #

21. # xinetd returns from sys_select. It is back on the runqueue and the scheduler makes it back running.

22. kernel_sched_schedule: 5277.041301367 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { prev_pid = 3689,
    next_pid = 4566, prev_state = 1 }

23. fs_select: 5277.041308682 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { fd = 3, timeout = -1 }

24. fs_select: 5277.041310786 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { fd = 5, timeout = -1 }

25. kernel_arch_syscall_exit: 5277.041314293 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret = 1 }

26. #

27. # xinetd accepts the connexion (sys_accept)

28. kernel_arch_syscall_entry: 5277.041321095 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id =
    102 [sys_socketcall+0x0/0x2d0], ip = 0xb8046424 }

29. net_socket_call: 5277.041322235 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { call = 5, a0 = 5 }

30. kernel_arch_syscall_exit: 5277.041333764 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret = 6 }

31. kernel_arch_syscall_entry: 5277.041342886 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id = 13
    [sys_time+0x0/0x30], ip = 0xb8046424 }

32. kernel_arch_syscall_exit: 5277.041344268 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret =
    1211549623 }

33. #

34. # xinetd carries on a fork(), the CPU is available for the process that was just created
```

```
35. kernel_arch_syscall_entry: 5277.041364003 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id =
    120 [sys_clone+0x0/0x40], ip = 0xb8046424 }

36. kernel_process_fork: 5277.041457157 (/tmp/trace-httpd/control/processes_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL {
    parent_pid = 4566, child_pid = 4979, child_tgid = 4979 }

37. kernel_sched_wakeup_new_task: 5277.041459577 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { pid = 4979,
    state = 0 }

38. kernel_sched_schedule: 5277.041466725 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { prev_pid = 4566,
    next_pid = 4979, prev_state = 0 }

39. kernel_arch_syscall_exit: 5277.041486672 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

40. #

41. # xinetd carries on the signal treatment setup for the new process

42. kernel_arch_syscall_entry: 5277.041836096 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    174 [sys_rt_sigaction+0x0/0xa0], ip = 0xb8046424 }

43. kernel_arch_syscall_exit: 5277.041840053 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

44. kernel_arch_syscall_entry: 5277.041842114 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    174 [sys_rt_sigaction+0x0/0xa0], ip = 0xb8046424 }

45. kernel_arch_syscall_exit: 5277.041843332 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

46. kernel_arch_syscall_entry: 5277.041845174 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    174 [sys_rt_sigaction+0x0/0xa0], ip = 0xb8046424 }

47. kernel_arch_syscall_exit: 5277.041846569 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

48. kernel_arch_syscall_entry: 5277.041848351 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    174 [sys_rt_sigaction+0x0/0xa0], ip = 0xb8046424 }

49. kernel_arch_syscall_exit: 5277.041849365 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

50. kernel_arch_syscall_entry: 5277.041865837 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    175 [sys_rt_sigprocmask+0x0/0x110], ip = 0xb8046424 }

51. kernel_arch_syscall_exit: 5277.041868008 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

52. #

53. # xinetd closes useless file descriptors (for “micro_httpd”)
```

```
54. kernel_arch_syscall_entry: 5277.041882743 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    6 [sys_close+0x0/0x110], ip = 0xb8046424 }

55. fs_close: 5277.041884668 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { fd = 3 }

56. kernel_arch_syscall_exit: 5277.041886529 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

57. kernel_arch_syscall_entry: 5277.041888033 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    6 [sys_close+0x0/0x110], ip = 0xb8046424 }

58. fs_close: 5277.041888832 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { fd = 4 }

59. kernel_arch_syscall_exit: 5277.041889729 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

60. kernel_arch_syscall_entry: 5277.041891157 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    6 [sys_close+0x0/0x110], ip = 0xb8046424 }

61. fs_close: 5277.041891919 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { fd = 0 }

62. kernel_arch_syscall_exit: 5277.041893175 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

63. kernel_arch_syscall_entry: 5277.041894615 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    6 [sys_close+0x0/0x110], ip = 0xb8046424 }

64. fs_close: 5277.041895357 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { fd = 1 }

65. kernel_arch_syscall_exit: 5277.041896043 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

66. kernel_arch_syscall_entry: 5277.041897466 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    6 [sys_close+0x0/0x110], ip = 0xb8046424 }

67. fs_close: 5277.041898192 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { fd = 2 }

68. #

69. # The process is currently “root”. Process’s UID and PID are replaced by “nobody”.

70. kernel_arch_syscall_exit: 5277.041898870 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

71. kernel_arch_syscall_entry: 5277.041928143 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    214 [sys_setgid+0x0/0x100], ip = 0xb8046424 }

72. kernel_arch_syscall_exit: 5277.041931343 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

73. kernel_arch_syscall_entry: 5277.041956887 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    206 [sys_setgroups+0x0/0x100], ip = 0xb8046424 }
```

```
74. kernel_arch_syscall_exit: 5277.041960633 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

75. kernel_arch_syscall_entry: 5277.041964819 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    213 [sys_setuid+0x0/0x150], ip = 0xb8046424 }

76. kernel_arch_syscall_exit: 5277.044378401 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

77. #

78. # getpeername()

79. kernel_arch_syscall_entry: 5277.044582578 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    102 [sys_socketcall+0x0/0x2d0], ip = 0xb8046424 }

80. net_socket_call: 5277.044584667 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { call = 7, a0 = 6 }

81. kernel_arch_syscall_exit: 5277.044588931 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

82. #

83. # getsockname()

84. kernel_arch_syscall_entry: 5277.044593305 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    102 [sys_socketcall+0x0/0x2d0], ip = 0xb8046424 }

85. net_socket_call: 5277.044594417 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { call = 6, a0 = 6 }

86. kernel_arch_syscall_exit: 5277.044595992 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

87. #

88. # Read the file “hosts.allow”; is the connexion should be accepted?

89. kernel_arch_syscall_entry: 5277.044662474 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    5 [sys_open+0x0/0x40], ip = 0xb8046424 }

90. fs_open: 5277.044684136 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { fd = 0, filename =
    "/etc/hosts.allow" }

91. kernel_arch_syscall_exit: 5277.044685328 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }

92. kernel_arch_syscall_entry: 5277.044736861 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    197 [sys_fstat64+0x0/0x30], ip = 0xb8046424 }

93. kernel_arch_syscall_exit: 5277.044740104 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0 }
```

```
94. kernel_arch_syscall_entry: 5277.044747184 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    192 [sys_mmap2+0x0/0xe0], ip = 0xb8046424 }

95. kernel_arch_syscall_exit: 5277.044755413 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = -
    1207676928 }

96. kernel_arch_syscall_entry: 5277.044757526 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { syscall_id =
    3 [sys_read+0x0/0xa0], ip = 0xb8046424 }

97. fs_read: 5277.044759137 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { fd = 0, count = 4096 }

98. kernel_timer_set: 5277.044781687 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { expires = 1488300,
    function = 0xe1037100, data = 3753167648 }

99. kernel_arch_syscall_exit: 5277.044800527 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 98 }

100.    kernel_arch_syscall_entry: 5277.044944978 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL {
    syscall_id = 6 [sys_close+0x0/0x110], ip = 0xb8046424 }

101.    fs_close: 5277.044946441 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { fd = 0 }

102.    kernel_arch_syscall_exit: 5277.044952676 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0
    }

103.    kernel_arch_syscall_entry: 5277.044954851 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL {
    syscall_id = 91 [sys_munmap+0x0/0x60], ip = 0xb8046424 }

104.    kernel_arch_syscall_exit: 5277.044967635 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0
    }

105.    kernel_arch_syscall_entry: 5277.045056131 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL {
    syscall_id = 221 [sys_fcntl64+0x0/0xc0], ip = 0xb8046424 }

106.    #

107.    # Close the listening socket (port 80).

108.    kernel_arch_syscall_exit: 5277.045059620 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0
    }

109.    kernel_arch_syscall_entry: 5277.045080411 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL {
    syscall_id = 6 [sys_close+0x0/0x110], ip = 0xb8046424 }

110.    fs_close: 5277.045081321 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { fd = 7 }

111.    #

112.    # Process's input/output redirection. (stdin, stdout et stderr toward the socket).
```

```
113.    kernel_arch_syscall_exit: 5277.045082970 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0
    }

114.    kernel_arch_syscall_entry:  5277.045084916  (/tmp/trace-httpd/cpu_0),  4979,  4979,  xinetd,  ,  4566,  0x0,  SYSCALL  {
    syscall_id = 63 [sys_dup2+0x0/0x120], ip = 0xb8046424 }

115.    kernel_arch_syscall_exit: 5277.045086316 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0
    }

116.    kernel_arch_syscall_entry:  5277.045087717  (/tmp/trace-httpd/cpu_0),  4979,  4979,  xinetd,  ,  4566,  0x0,  SYSCALL  {
    syscall_id = 63 [sys_dup2+0x0/0x120], ip = 0xb8046424 }

117.    kernel_arch_syscall_exit: 5277.045088524 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 1
    }

118.    kernel_arch_syscall_entry:  5277.045089876  (/tmp/trace-httpd/cpu_0),  4979,  4979,  xinetd,  ,  4566,  0x0,  SYSCALL  {
    syscall_id = 63 [sys_dup2+0x0/0x120], ip = 0xb8046424 }

119.    kernel_arch_syscall_exit: 5277.045090680 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 2
    }

120.    #

121.    # Impose a resource limit.

122.    kernel_arch_syscall_entry:  5277.045098086  (/tmp/trace-httpd/cpu_0),  4979,  4979,  xinetd,  ,  4566,  0x0,  SYSCALL  {
    syscall_id = 75 [sys_setrlimit+0x0/0x210], ip = 0xb8046424 }

123.    kernel_arch_syscall_exit: 5277.045100301 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0
    }

124.    #

125.    # Close the original socket file descriptor (socket's redirection toward stdin/stdout was already done)

126.    kernel_arch_syscall_entry:  5277.045102124  (/tmp/trace-httpd/cpu_0),  4979,  4979,  xinetd,  ,  4566,  0x0,  SYSCALL  {
    syscall_id = 6 [sys_close+0x0/0x110], ip = 0xb8046424 }

127.    fs_close: 5277.045102888 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { fd = 6 }

128.    kernel_arch_syscall_exit: 5277.045103944 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret = 0
    }

129.    #

130.    # Launch "micro_httpd"
```

```
131.    kernel_arch_syscall_entry:  5277.045111538  (/tmp/trace-httpd/cpu_0),  4979,  4979,  xinetd,  ,  4566,  0x0,  SYSCALL  {
    syscall_id = 66 [sys_setsid+0x0/0xd0], ip = 0xb8046424 }

132.    kernel_arch_syscall_exit: 5277.045115067 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, USER_MODE { ret =
    4979 }

133.    kernel_arch_syscall_entry:  5277.045125131  (/tmp/trace-httpd/cpu_0),  4979,  4979,  xinetd,  ,  4566,  0x0,  SYSCALL  {
    syscall_id = 11 [sys_execve+0x0/0x80], ip = 0xb8046424 }

134.    fs_close: 5277.045319969 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { fd = 5 }

135.    fs_close: 5277.045394854 (/tmp/trace-httpd/cpu_0), 4979, 4979, xinetd, , 4566, 0x0, SYSCALL { fd = 3 }

136.    fs_exec: 5277.045421309 (/tmp/trace-httpd/control/processes_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0,
    SYSCALL { filename = "/usr/local/sbin/micro_httpd" }

137.    kernel_arch_syscall_exit:  5277.045424721  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

138.    #

139.    # Initialize “micro_httpd”. Load libraries (.so).

140.    kernel_arch_syscall_entry:  5277.045473574  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 45 [sys_brk+0x0/0x100], ip = 0xb805becb }

141.    kernel_arch_syscall_exit:  5277.045474938  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 134524928 }

142.    kernel_arch_syscall_entry:  5277.045553130  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 33 [sys_access+0x0/0x30], ip = 0xb805c761 }

143.    kernel_arch_syscall_exit:  5277.045561376  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = -2 }

144.    kernel_arch_syscall_entry:  5277.045597090  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 5 [sys_open+0x0/0x40], ip = 0xb805c624 }

145.    fs_open: 5277.045610405 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd =
    3, filename = "/etc/ld.so.cache" }

146.    kernel_arch_syscall_exit:  5277.045611506  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 3 }

147.    kernel_arch_syscall_entry:  5277.045613448  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 197 [sys_fstat64+0x0/0x30], ip = 0xb805c5ee }

148.    kernel_arch_syscall_exit:  5277.045616316  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }
```

```
149.    kernel_arch_syscall_entry:  5277.045618160  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 192 [sys_mmap2+0x0/0xe0], ip = 0xb805c8b3 }

150.    kernel_arch_syscall_exit:  5277.045637317  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = -1207734272 }

151.    kernel_arch_syscall_entry:  5277.045639201  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 6 [sys_close+0x0/0x110], ip = 0xb805c65d }

152.    fs_close: 5277.045640113 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd =
    3 }

153.    kernel_arch_syscall_exit:  5277.045641698  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

154.    kernel_arch_syscall_entry:  5277.045699289  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 5 [sys_open+0x0/0x40], ip = 0xb805c624 }

155.    fs_open: 5277.045711045 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd =
    3, filename = "/lib/libc.so.6" }

156.    kernel_arch_syscall_exit:  5277.045712189  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 3 }

157.    kernel_arch_syscall_entry:  5277.045714048  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 3 [sys_read+0x0/0xa0], ip = 0xb805c6a4 }

158.    fs_read: 5277.045715671 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd =
    3, count = 512 }

159.    kernel_arch_syscall_exit:  5277.045723748  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 512 }

160.    kernel_arch_syscall_entry:  5277.045733316  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 197 [sys_fstat64+0x0/0x30], ip = 0xb805c5ee }

161.    kernel_arch_syscall_exit:  5277.045734639  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

162.    kernel_arch_syscall_entry:  5277.045737333  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 192 [sys_mmap2+0x0/0xe0], ip = 0xb805c8b3 }

163.    kernel_arch_syscall_exit:  5277.045740834  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = -1207738368 }

164.    kernel_arch_syscall_entry:  5277.045763122  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 192 [sys_mmap2+0x0/0xe0], ip = 0xb805c8b3 }

165.    kernel_arch_syscall_exit:  5277.045766827  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = -1209049088 }
```

```
166.    kernel_arch_syscall_entry:  5277.045768633  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 192 [sys_mmap2+0x0/0xe0], ip = 0xb805c8b3 }

167.    kernel_arch_syscall_exit:  5277.045781522  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = -1207762944 }

168.    kernel_arch_syscall_entry:  5277.045799596  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 192 [sys_mmap2+0x0/0xe0], ip = 0xb805c8b3 }

169.    kernel_arch_syscall_exit:  5277.045804336  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = -1207750656 }

170.    kernel_arch_syscall_entry:  5277.045826075  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 6 [sys_close+0x0/0x110], ip = 0xb805c65d }

171.    fs_close: 5277.045826953 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd =
    3 }

172.    kernel_arch_syscall_exit:  5277.045827976  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

173.    kernel_arch_syscall_entry:  5277.045878785  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 192 [sys_mmap2+0x0/0xe0], ip = 0xb805c8b3 }

174.    kernel_arch_syscall_exit:  5277.045882269  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = -1209053184 }

175.    kernel_arch_syscall_entry:  5277.045897587  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 243 [sys_set_thread_area+0x0/0x30], ip = 0xb8049180 }

176.    kernel_arch_syscall_exit:  5277.045900276  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

177.    kernel_arch_syscall_entry:  5277.046140617  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 125 [sys_mprotect+0x0/0x240], ip = 0xb805c934 }

178.    kernel_arch_syscall_exit:  5277.046150040  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

179.    kernel_arch_syscall_entry:  5277.046162358  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 91 [sys_munmap+0x0/0x60], ip = 0xb805c8f1 }

180.    kernel_arch_syscall_exit:  5277.046178697  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

181.    #

182.    # micro_httpd carries on a chdir() in order to move into the directory containing server's files
```

```
183.    kernel_arch_syscall_entry:  5277.046274494  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 12 [sys_chdir+0x0/0x70], ip = 0xb8046424 }

184.    kernel_arch_syscall_exit:  5277.046283568  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

185.    #

186.    # Read stdin (HTTP request through the socket)

187.    kernel_arch_syscall_entry:  5277.046327583  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 197 [sys_fstat64+0x0/0x30], ip = 0xb8046424 }

188.    kernel_arch_syscall_exit:  5277.046331186  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

189.    kernel_arch_syscall_entry:  5277.046339051  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 192 [sys_mmap2+0x0/0xe0], ip = 0xb8046424 }

190.    kernel_arch_syscall_exit:  5277.046343087  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = -1207676928 }

191.    kernel_arch_syscall_entry:  5277.046345355  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 3 [sys_read+0x0/0xa0], ip = 0xb8046424 }

192.    fs_read: 5277.046346592 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd =
    0, count = 1024 }

193.    kernel_arch_syscall_exit:  5277.046372029  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 405 }

194.    #

195.    # fstat() is triggered (probably on the "index.html" file)

196.    kernel_arch_syscall_entry:  5277.046513347  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 195 [sys_stat64+0x0/0x30], ip = 0xb8046424 }

197.    kernel_arch_syscall_exit:  5277.046519411  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

198.    #

199.    # The Advance Programmable Interrupt Controller (APIC) sends a signal to the CPU

200.    kernel_irq_entry: 5277.046591673 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, IRQ {
    irq_id = 239, kernel_mode = 0 }
```

```
201.    kernel_timer_update_time:  5277.046601946  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0,  IRQ  {  jiffies  =  4296453959,  xtime_sec  =  1211549623,  xtime_nsec  =  519962424,  walltomonotonic_sec  =  -1211544369,
    walltomonotonic_nsec = 753353119 }

202.    kernel_softirq_raise:  5277.046605090  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,  0x0,
    IRQ { softirq_id = 1 [run_timer_softirq+0x0/0x1f0] }

203.    kernel_softirq_raise:  5277.046608207  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,  0x0,
    IRQ { softirq_id = 8 [rcu_process_callbacks+0x0/0x30] }

204.    kernel_irq_exit:  5277.046615125  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,  0x0,
    USER_MODE

205.    kernel_softirq_entry:  5277.046616638  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,  0x0,
    SOFTIRQ { softirq_id = 1 [run_timer_softirq+0x0/0x1f0] }

206.    kernel_timer_set:  5277.046624617  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,  0x0,
    SOFTIRQ { expires = 1486664, function = 0xc024f950, data = 0 }

207.    kernel_softirq_exit:  5277.046625807  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,  0x0,
    USER_MODE { softirq_id = 1 [run_timer_softirq+0x0/0x1f0] }

208.    kernel_softirq_entry:  5277.046626514  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,  0x0,
    SOFTIRQ { softirq_id = 8 [rcu_process_callbacks+0x0/0x30] }

209.    kernel_softirq_exit:  5277.046628392  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,  0x0,
    USER_MODE { softirq_id = 8 [rcu_process_callbacks+0x0/0x30] }

210.    kernel_arch_syscall_entry:  5277.046711024  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 45 [sys_brk+0x0/0x100], ip = 0xb8046424 }

211.    kernel_arch_syscall_exit:  5277.046712674  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 134524928 }

212.    kernel_arch_syscall_entry:  5277.046714192  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 45 [sys_brk+0x0/0x100], ip = 0xb8046424 }

213.    kernel_arch_syscall_exit:  5277.046719018  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 134660096 }

214.    #

215.    # Open the file “index.html”

216.    kernel_arch_syscall_entry:  5277.046748424  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 5 [sys_open+0x0/0x40], ip = 0xb8046424 }

217.    fs_open: 5277.046759271 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd =
    3, filename = "index.html" }
```

```
218.    kernel_arch_syscall_exit:  5277.046760425  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 3 }

219.    kernel_arch_syscall_entry:  5277.046814182  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 197 [sys_fstat64+0x0/0x30], ip = 0xb8046424 }

220.    kernel_arch_syscall_exit:  5277.046816916  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

221.    kernel_arch_syscall_entry:  5277.046818594  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 192 [sys_mmap2+0x0/0xe0], ip = 0xb8046424 }

222.    kernel_arch_syscall_exit:  5277.046823194  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = -1207681024 }

223.    #

224.    # Read the system time (to be inserted in the HTTP header; see send_header())

225.    # (glibc opens "/etc/localtime" to know the system's time zone)

226.    kernel_arch_syscall_entry:  5277.046854659  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 13 [sys_time+0x0/0x30], ip = 0xb8046424 }

227.    kernel_arch_syscall_exit:  5277.046856255  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 1211549623 }

228.    kernel_arch_syscall_entry:  5277.046888105  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 5 [sys_open+0x0/0x40], ip = 0xb8046424 }

229.    fs_open: 5277.046899066 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd =
    4, filename = "/etc/localtime" }

230.    kernel_arch_syscall_exit:  5277.046900144  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 4 }

231.    kernel_arch_syscall_entry:  5277.046908152  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 197 [sys_fstat64+0x0/0x30], ip = 0xb8046424 }

232.    kernel_arch_syscall_exit:  5277.046909877  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

233.    kernel_arch_syscall_entry:  5277.046918051  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 197 [sys_fstat64+0x0/0x30], ip = 0xb8046424 }

234.    kernel_arch_syscall_exit:  5277.046919215  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }
```

```
235.    kernel_arch_syscall_entry:  5277.046920681  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 192 [sys_mmap2+0x0/0xe0], ip = 0xb8046424 }

236.    kernel_arch_syscall_exit:  5277.046922723  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = -1207685120 }

237.    kernel_arch_syscall_entry:  5277.046925133  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 3 [sys_read+0x0/0xa0], ip = 0xb8046424 }

238.    fs_read: 5277.046926752 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd =
    4, count = 4096 }

239.    kernel_arch_syscall_exit:  5277.046964889  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 3477 }

240.    kernel_arch_syscall_entry:  5277.046988062  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 140 [sys_llseek+0x0/0xd0], ip = 0xb8046424 }

241.    fs_llseek: 5277.046990756 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd
    = 4, offset = 3453, origin = 1 }

242.    kernel_arch_syscall_exit:  5277.046991972  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

243.    kernel_arch_syscall_entry:  5277.047000434  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 3 [sys_read+0x0/0xa0], ip = 0xb8046424 }

244.    fs_read: 5277.047001351 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd =
    4, count = 4096 }

245.    kernel_arch_syscall_exit:  5277.047004344  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 24 }

246.    kernel_arch_syscall_entry:  5277.047008992  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 6 [sys_close+0x0/0x110], ip = 0xb8046424 }

247.    fs_close: 5277.047010329 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd =
    4 }

248.    kernel_arch_syscall_exit:  5277.047015507  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

249.    kernel_arch_syscall_entry:  5277.047017573  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 91 [sys_munmap+0x0/0x60], ip = 0xb8046424 }

250.    kernel_arch_syscall_exit:  5277.047034426  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

251.    kernel_arch_syscall_entry:  5277.047080847  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 195 [sys_stat64+0x0/0x30], ip = 0xb8046424 }
```

```
252.    kernel_arch_syscall_exit:  5277.047087277  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

253.    kernel_arch_syscall_entry:  5277.047115531  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 195 [sys_stat64+0x0/0x30], ip = 0xb8046424 }

254.    kernel_arch_syscall_exit:  5277.047118841  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

255.    #

256.    # Read the file "index.html"

257.    kernel_arch_syscall_entry:  5277.047128063  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 197 [sys_fstat64+0x0/0x30], ip = 0xb8046424 }

258.    kernel_arch_syscall_exit:  5277.047130031  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

259.    kernel_arch_syscall_entry:  5277.047131975  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 192 [sys_mmap2+0x0/0xe0], ip = 0xb8046424 }

260.    kernel_arch_syscall_exit:  5277.047135197  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = -1207685120 }

261.    kernel_arch_syscall_entry:  5277.047137101  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 3 [sys_read+0x0/0xa0], ip = 0xb8046424 }

262.    fs_read: 5277.047138264 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd =
    3, count = 4096 }

263.    kernel_arch_syscall_exit:  5277.047158913  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 103 }

264.    kernel_arch_syscall_entry:  5277.047170662  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, SYSCALL { syscall_id = 3 [sys_read+0x0/0xa0], ip = 0xb8046424 }

265.    fs_read: 5277.047171721 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd =
    3, count = 4096 }

266.    kernel_arch_syscall_exit:  5277.047173613  (/tmp/trace-httpd/cpu_0),  4979,  4979,  /usr/local/sbin/micro_httpd,  ,  4566,
    0x0, USER_MODE { ret = 0 }

267.    #

268.    # Flush stdout. The answer is sent on the network.

269.    # As the test is done on the loopback (127.0.0.1), the HTTP client's endpoint receives the packet.
```

```
# The ACK is retransmitted immediately by the endpoint.
kernel_arch_syscall_entry: 5277.047178396 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { syscall_id = 4 [sys_write+0x0/0xa0], ip = 0xb8046424 }
fs_write: 5277.047179914 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { fd = 1, count = 311 }
net_dev_xmit: 5277.047200820 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { skb = 0xd96c9cb4, protocol = 8 }
kernel_softirq_entry: 5277.047206305 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SOFTIRQ { softirq_id = 3 [net_rx_action+0x0/0x240] }
net_dev_receive: 5277.047209489 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SOFTIRQ { skb = 0xd96c9cb4, protocol = 8 }
net_dev_xmit: 5277.047224538 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SOFTIRQ { skb = 0xd96ade00, protocol = 8 }
kernel_sched_try_wakeup: 5277.047228132 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SOFTIRQ { pid = 3689, state = 1 }
net_dev_receive: 5277.047235954 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SOFTIRQ { skb = 0xd96ade00, protocol = 8 }
kernel_softirq_exit: 5277.047238183 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { softirq_id = 3 [net_rx_action+0x0/0x240] }
kernel_timer_set: 5277.047241141 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { expires = 1486724, function = 0xc03336b0, data = 3745083136 }
kernel_arch_syscall_exit: 5277.047252314 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, USER_MODE { ret = 311 }
#
# End the process (exit() is called)
kernel_arch_syscall_entry: 5277.047281311 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { syscall_id = 252 [sys_exit_group+0x0/0x20], ip = 0xb8046424 }
kernel_process_exit: 5277.047374690 (/tmp/trace-httpd/control/processes_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0, SYSCALL { pid = 4979 }
#
# SIGCHD is sent to xinetd (micro_httpd has finished). The CPU resource is given back to xinetd.
```

```
288.    kernel_send_signal: 5277.047388531 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0,
    SYSCALL { pid = 4566, signal = 17 }

289.    kernel_sched_try_wakeup: 5277.047391850 (/tmp/trace-httpd/cpu_0), 4979, 4979, /usr/local/sbin/micro_httpd, , 4566, 0x0,
    SYSCALL { pid = 4566, state = 0 }

290.    kernel_sched_schedule: 5277.070484305 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { prev_pid =
    3318, next_pid = 4566, prev_state = 1 }

291.    kernel_arch_syscall_exit: 5277.070492080 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret = 4979
    }

292.    #

293.    # Reaction to the SIGCHD signal (within the signal handler)

294.    # Write in a pipe (probably toward syslogd)

295.    kernel_arch_syscall_entry: 5277.070517471 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id
    = 4 [sys_write+0x0/0xa0], ip = 0xb8046424 }

296.    fs_write: 5277.070519201 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { fd = 4, count = 1 }

297.    kernel_arch_syscall_exit: 5277.070527602 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret = 1 }

298.    kernel_arch_syscall_entry: 5277.070535745 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id
    = 119 [sys_sigreturn+0x0/0x180], ip = 0xb8046408 }

299.    kernel_arch_syscall_exit: 5277.070537570 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret = 4979
    }

300.    #

301.    # Back to the normal execution of xinetd (not anymore in the signal handler)

302.    kernel_arch_syscall_entry: 5277.070565577 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id
    = 13 [sys_time+0x0/0x30], ip = 0xb8046424 }

303.    kernel_arch_syscall_exit: 5277.070566610 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret =
    1211549623 }

304.    kernel_arch_syscall_entry: 5277.070637518 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id
    = 13 [sys_time+0x0/0x30], ip = 0xb8046424 }

305.    kernel_arch_syscall_exit: 5277.070638493 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret =
    1211549623 }
```

```
kernel_arch_syscall_entry: 5277.070662450 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id = 195 [sys_stat64+0x0/0x30], ip = 0xb8046424 }

kernel_arch_syscall_exit: 5277.070677035 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret = 0 }

#

# Write on the socket

kernel_arch_syscall_entry: 5277.070696689 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id = 102 [sys_socketcall+0x0/0x2d0], ip = 0xb8046424 }

net_socket_call: 5277.070698950 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { call = 9, a0 = 7 }

net_socket_sendmsg: 5277.070702810 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { sock = 0xda9afa00, family = 1, type = 1, protocol = 0, size = 70 }

#

# syslogd is awakened (syslog trace is omitted in this source)

kernel_sched_try_wakeup: 5277.070711591 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { pid = 3163, state = 1 }

kernel_sched_schedule: 5277.071041070 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { prev_pid = 3163, next_pid = 4566, prev_state = 1 }

#

# Back to xinitd

kernel_arch_syscall_exit: 5277.071047862 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret = 70 }

#

# Close the socket

# Client's endpoint receives the FIN and returns an ACK

kernel_arch_syscall_entry: 5277.071054615 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id = 6 [sys_close+0x0/0x110], ip = 0xb8046424 }

fs_close: 5277.071056331 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { fd = 6 }

net_dev_xmit: 5277.071078320 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { skb = 0xd96c90b4, protocol = 8 }
```

```
326.    kernel_softirq_entry: 5277.071084062 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SOFTIRQ { softirq_id = 3
    [net_rx_action+0x0/0x240] }

327.    net_dev_receive: 5277.071086768 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SOFTIRQ { skb = 0xd96c90b4,
    protocol = 8 }

328.    kernel_timer_set: 5277.071104161 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SOFTIRQ { expires = 1504670,
    function = 0xc0333170, data = 3745084416 }

329.    net_dev_xmit: 5277.071109261 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SOFTIRQ { skb = 0xd96add00,
    protocol = 8 }

330.    kernel_timer_set: 5277.071113756 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SOFTIRQ { expires = 1488920,
    function = 0xc0321ae0, data = 3225789568 }

331.    net_dev_receive: 5277.071122082 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SOFTIRQ { skb = 0xd96add00,
    protocol = 8 }

332.    kernel_softirq_exit: 5277.071124317 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { softirq_id = 3
    [net_rx_action+0x0/0x240] }

333.    kernel_timer_set: 5277.071125747 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { expires = 1486731,
    function = 0xc03336b0, data = 3745083136 }

334.    kernel_arch_syscall_exit: 5277.071140152 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret = 0 }

335.    #

336.    # xinetd listen on file descriptors

337.    kernel_arch_syscall_entry: 5277.071157392 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id
    = 142 [sys_select+0x0/0x1c0], ip = 0xb8046424 }

338.    fs_select: 5277.071161262 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { fd = 3, timeout = -1 }

339.    fs_select: 5277.071163841 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { fd = 5, timeout = -1 }

340.    kernel_arch_syscall_exit: 5277.071166761 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret = 1 }

341.    #

342.    # Read on the pipe

343.    kernel_arch_syscall_entry: 5277.071169693 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id
    = 54 [sys_ioctl+0x0/0xb0], ip = 0xb8046424 }

344.    fs_ioctl: 5277.071171073 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { fd = 3, cmd = 21531, arg =
    3213234188 }
```

```
345.    kernel_arch_syscall_exit: 5277.071173801 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret = 0 }

346.    kernel_arch_syscall_entry: 5277.071175481 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id
    = 3 [sys_read+0x0/0xa0], ip = 0xb8046424 }

347.    fs_read: 5277.071176811 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { fd = 3, count = 1 }

348.    kernel_arch_syscall_exit: 5277.071184314 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret = 1 }

349.    #

350.    # Get the status of the child process

351.    kernel_arch_syscall_entry: 5277.071187101 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id
    = 7 [sys_waitpid+0x0/0x30], ip = 0xb8046424 }

352.    kernel_process_wait: 5277.071189157 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { pid = 0 }

353.    kernel_arch_syscall_exit: 5277.071204254 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret = 4979
    }

354.    #

355.    # Close the file descriptor

356.    # This call fails (error EBADF ; Bad file number). The file was probably already closed.

357.    kernel_arch_syscall_entry: 5277.071207890 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id
    = 6 [sys_close+0x0/0x110], ip = 0xb8046424 }

358.    kernel_arch_syscall_exit: 5277.071209118 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret = -9 }

359.    #

360.    # A second waitpid().

361.    # It fails (error ECHILD ; No child process)

362.    kernel_arch_syscall_entry: 5277.071212901 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id
    = 7 [sys_waitpid+0x0/0x30], ip = 0xb8046424 }

363.    kernel_process_wait: 5277.071213687 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { pid = 0 }

364.    kernel_arch_syscall_exit: 5277.071214958 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, USER_MODE { ret = -10
    }

365.    #
```

```
366.    # Finally xinetd is back waiting
367.    kernel_arch_syscall_entry: 5277.071217266 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { syscall_id
    = 142 [sys_select+0x0/0x1c0], ip = 0xb8046424 }
368.    fs_select: 5277.071219082 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { fd = 3, timeout = -1 }
369.    fs_select: 5277.071220157 (/tmp/trace-httpd/cpu_0), 4566, 4566, xinetd, , 1, 0x0, SYSCALL { fd = 5, timeout = -1 }
370.    #
371.    # End of the trace.
```

This page intentionally left blank.

# Annex F Glossary

This glossary contains definitions of some selected terms from this document. It does not claim to: 1) be complete or 2) reflect the work of all researchers in this domain.

**Availability (survivability)**

Avizienis et al. (2004) mention: *it is the readiness for correct service*. Neumann (2000) adds: *availability implies that certain required resources are available when and as needed. It can be applied at many levels of abstraction, including systems, subsystems, data entities, and communication links. Prevention of denial of service is an availability requirements, although it also has a system-integrity component.*

- **Data availability**: Preventing disruptions in timely access to data, including sensor data in a control system. Multiple versions of critical data and alternative sensors can help increase data availability.
- **Network availability**: Detecting, preventing, and recovering from denial-of-service attacks, such as outages of network nodes and access devices, electromagnetic interference on the communications media.
- **Real-time availability**: (Including the system, data, and other resources). Ensuring that real-time processing can be done in a timely way, protecting against maliciously or accidentally caused delays.
- **System availability**: Preventing system and communication outages, and even temporary unavailability of resources. Such outages may include malicious or accidental denials of system service.

**Correctness or functional correctness (survivability)**

*Assuring that a flaw in the application or in the computer operating system, or a human error in system maintenance, cannot compromise the application* (Neumann, 2000).

**Dependability (survivability)**

Avizienis et al. (2004) propose two definitions for dependability. *The first one stresses the need for justification of trust: the ability to deliver service that can justifiably be trusted. The second one provides the criterion for deciding if the service is dependable: the ability to avoid service failures that are more frequent and more severe than is acceptable.* Neumann (2000) mentions: *In the fault-tolerance community, dependability tends to be a measure of how well the specified fault-tolerance requirements are met, although recent usage is generalizing that to other requirements. It may be considered as the extent to which a given survivability requirement is perceived to be satisfied, particularly by the implementation.*

**Error (survivability)**

*When at least one (or more) external state of the system deviates from the correct service state* Avizienis (2004). The definition of an error is the part of the total state of the system that may lead to its subsequent service failure.

**Failure (survivability)**

see Service failure.

**Fault (survivability)**

*The adjudged or hypothesized cause of an error is called a fault* (Avizienis, 2004). The following definitions were extracted form the same paper.

- ***Commission faults****: Performing wrong actions leads to commission faults.*
- ***Configuration faults****: Wrong setting of parameters that can affect security, networking, storage, middleware, etc.*
- ***Deliberate faults****: Faults that are due to bad decisions, that is, intended actions that are wrong and cause faults.*
- ***Development faults****: Includes all faults classes occuring during development.*
- ***Human-made faults****: Faults that result from human actions.*
- ***Interaction faults****: Include all external faults. (...) a common feature of interaction faults is that, in order to be "successful," they usually necessitate the prior presence of a vulnerability, i.e. an internal fault that enables an external faults to harm the system*
- ***Intermittent faults****:*
- ***Malicious faults****: Introduced during either system development with the objective to cause harm to the system during its use, or directly during use.*
- ***Multiple faults****: Multiple faults are two or more concurrent, overlapping, or sequential single faults whose consequences, i.e., errors, overlap in time, that is, the errors due to these faults are concurrently present in the system. Consideration of multiple faults leads one to distinguish 1) independent faults that are attributed to different causes and 2) related faults, that are attributed to a common cause. Related faults generally cause similar errors, i.e., errors that cannot be distinguished by whatever detection mechanisms are being employed, whereas independent faults usually cause distinct errors.*
- ***Natural faults****: Natural fault are physical (hardware) faults that are caused by natural phenomena without human participation.*
- ***Non-deliberate faults****: Faults that are due to mistakes, that is, unintended actions of which the developer, operator, maintainer, etc. is not aware.*
- ***Non-malicious faults****: Introduced without malicious objectives.*
- ***Omission faults****: Refers to faults that result when no action is performed while action should be performed.*
- ***Physical faults****: Includes all fault classes that affect hardware.*

- ***Single faults****: Single fault is a fault caused by one adverse physical event or one harmful human action.*

**Fault tolerance (survivability)**

*Preventing undesirable effects resulting from failures of underlying hardware components, subsystems, or indeed the entire system* (Neumann, 2000). *To avoid service failures in the presence of faults.* (...) Fault tolerance [3], *which is aimed at failure avoidance, is carried out via error detection and system recovery.* (...) *Fault tolerance applies to all classes of faults.* (...) s*everal synonyms exist for fault tolerance: self-repair, self-healing, resilience.* (...) (Avizienis, 2004). (…)

- ***Error Detection****: Identifies the presence of an error.*
- ***Concurrent detection****: Takes place during normal service delivery.*
- ***Preemptive detection****: Takes place while normal service delivery is suspended; checks the system for latent errors and dormant faults.*
- ***Recovery****: Transforms a system state that contains one or more errors and (possibly) faults into a state without detected errors and without faults that can be activated again.}*
- ***Error handling****: Eliminates errors from the system state.*
- ***Rollback****: Brings the system back to a saved state that existed prior to error occurence; saved state; checkpoint.*
- ***Rollforward****: State without detected errors is a new state.*
- ***Compensation****: The erroneous contains enough redundancy to enable error to be masked.*
- ***Fault handling****: Prevents faults from being activated again.*
- ***Diagnosis****: Identifies and records the cause(s) of error(s), in terms of both location and state.*
- ***Isolation****: Performs physical or logical exclusion of the faulty components from further participation in service delivery, i.e. makes the fault dormant.*
- ***Reconfiguration****: Either switches in space components or reassigns tasks among non-falled components.*
- ***Reinitialization****: Checks, updates and records the new configuration and updates system tables and records.*

**Ioctl (Linux)**

WikipediaWS (2008): *In computing, an ioctl (pronounced /aɪ□ɒktəl/ or "i-o-control") is part of the user-to-kernel interface of a conventional operating system. Short for "Input/output control", ioctls are typically employed to allow userspace code to communicate with hardware devices or kernel components.*

**Kernel (Linux)**

WikipediaWS (2008): *In computer science, the kernel is the central component of most computer operating systems (OS). Its responsibilities include managing the system's resources (the communication between hardware and software components).[1] As a basic component of an operating system, a kernel provides the lowest-level abstraction layer for the resources (especially memory, processors and I/O devices) that application software must control to perform its function. It typically makes these facilities available to application processes through inter-process communication mechanisms and system calls. These tasks are done differently by different kernels, depending on their design and implementation. While monolithic kernels will try to achieve these goals by executing all the code in the same address space to increase the performance of the system, microkernels run most of their services in user space, aiming to improve maintainability and modularity of the codebase.[2] A range of possibilities exists between these two extremes.*

**Kernel-context and user-space (Linux)**

WikipediaWS (2008): *A conventional operating system usually segregates virtual memory into kernel space and user space. Kernel space is strictly reserved for running the kernel, kernel extensions, and some device drivers. In most operating systems, kernel memory is never swapped out to disk. In contrast, user space is the memory area where all user mode applications work and this memory can be swapped out when necessary. The term userland is often used for referring to operating system software that runs in user space. Each user space process normally runs in its own virtual memory space, and, unless explicitly requested, cannot access the memory of other processes. This is the basis for memory protection in today's mainstream operating systems, and a building block for privilege separation. Depending on the privileges, processes can request the kernel to map part of another process's memory space to its own, as is the case for debuggers. Programs can also request shared memory regions with other processes. Another approach taken in experimental operating systems, is to have a single address space for all software, and rely on the programming language's virtual machine to make sure that arbitrary memory cannot be accessed — applications simply cannot acquire any references to the objects that they are not allowed to access.[1] This approach has been implemented in JXOS, Unununium as well as Microsoft's Singularity research project.*

**Multi-core CPU**

WikipediaWS (2008): *A multi-core CPU (or chip-level multiprocessor, CMP) combines two or more independent cores into a single package composed of a single integrated circuit (IC), called a die, or more dies packaged together. A dual-core processor contains two cores and a quad-core processor contains four cores. A multi-core microprocessor implements multiprocessing in a single physical package. A processor with all cores on a single die is called a monolithic processor. Cores in a multicore device may share a single coherent cache at the highest on-device cache level (e.g. L2 for the Intel Core 2) or may have separate caches (e.g. current AMD dual-core processors). The processors also share the same interconnect to the rest of the system. Each "core" independently implements optimizations such as superscalar execution, pipelining, and multithreading. A system with N cores is effective when it is presented with N or more threads concurrently. The most commercially significant (or at least the most 'obvious') multi-core processors are those used in computers (primarily from Intel & AMD) and game consoles (e.g., the Cell processor in the PS3). In this context, "multi" typically means a relatively small number of cores. However, the technology is widely used in other technology areas, especially*

*those of embedded processors, such as network processors and digital signal processors, and in GPUs.*

**Multi-processing, multi-processors**

WikipediaWS (2008): *Multiprocessing is the use of two or more central processing units (CPUs) within a single computer system. The term also refers to the ability of a system to support more than one processor and/or the ability to allocate tasks between them.[1] There are many variations on this basic theme, and the definition of multiprocessing can vary with context, mostly as a function of how CPUs are defined (multiple cores on one die, multiple chips in one package, multiple packages in one system unit, etc.). Multiprocessing sometimes refers to the execution of multiple concurrent software processes in a system as opposed to a single process at any one instant. However, the term multiprogramming is more appropriate to describe this concept, which is implemented mostly in software, whereas multiprocessing is more appropriate to describe the use of multiple hardware CPUs. A system can be both multiprocessing and multiprogramming, only one of the two, or neither of the two.*

**Multi-tasking**

WikipediaWS (2008): *In computing, multitasking is a method by which multiple tasks, also known as processes, share common processing resources such as a CPU. In the case of a computer with a single CPU, only one task is said to be running at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called a context switch. When context switches occur frequently enough the illusion of parallelism is achieved. Even on computers with more than one CPU (called multiprocessor machines), multitasking allows many more tasks to be run than there are CPUs.*

**Netlink (Linux)**

WikipediaWS (2008): *Netlink is socket-like mechanism for IPC between kernel and user space processes, as well as between user-space processes alone (like e.g., unix sockets) or mixture of multiple user space and kernel space processes. However, unlike INET sockets, it can't traverse host boundaries, as it addresses processes by their (inherently local) PIDs. It was designed and is used to transfer miscellaneous networking information between the Linux kernel space and user space processes. Many networking utilities use netlink to communicate with linux kernel from user space, for example iproute2. Netlink consists of a standard socket-based interface for userspace process and an internal kernel API for kernel modules. It is designed to be a more flexible successor to ioctl. Originally netlink uses AF_NETLINK socket family.*

**Non-uniform memory access (NUMA)**

WikipediaWS (2008): *Non-Uniform Memory Access or Non-Uniform Memory Architecture (NUMA) is a computer memory design used in multiprocessors, where the memory access time depends on the memory location relative to a processor. Under NUMA, a processor can access its own local memory faster than non-local memory, that is, memory local to another processor or memory shared between processors. NUMA architectures logically follow in scaling from*

*symmetric multiprocessing (SMP) architectures. Their commercial development came in work by Burroughs, Convex Computer (later HP), SGI, Sequent and Data General during the 1990s. Techniques developed by these companies later featured in a variety of Unix-like operating systems, as well as to some degree in Windows NT.*

**Open Source or Free Open Source Software (FOSS)**

WikipediaWS (2008): *Open source is a set of principles and practices on how to write software, the most important of which is that the source code is openly available. The Open Source Definition, which was created by Bruce Perens[1] and Eric Raymond and is currently maintained by the Open Source Initiative, adds additional meaning to the term: one should not only get the source code but also have the right to use it. If the latter is denied the license is categorized as a shared source license.*

Foldoc (2008): *A method and philosophy for software licensing and distribution designed to encourage use and improvement of software written by volunteers by ensuring that anyone can copy the source code and modify it freely. The term "open source" is now more widely used than the earlier term "free software" (promoted by the Free Software Foundation) but has broadly the same meaning - free of distribution restrictions, not necessarily free of charge. There are various open source licenses available. Programmers can choose an appropriate license to use when distributing their programs. The Open Source Initiative promotes the Open Source Definition.*

**procfs (Linux)**

WikipediaWS (2008): *On many Unix-like computer systems, procfs, short for process file system, consists of a pseudo file system (a file system dynamically generated at boot) used to access process information from the kernel. The file system is often mounted at the /proc directory. Because /proc is not a real file system, it consumes no storage space and only a limited amount of memory. The following operating environments support procfs: Solaris; BSD; Linux (which extends it to non-process-related data); IBM AIX (operating system) (which bases its implementation on Linux to improve compatibility); QNX; Plan 9 from Bell Labs (where it originated).*

**Process**

WikipediaWS (2008): *In computing, a process is an instance of a computer program that is being sequentially executed.[1] A computer program itself is just a passive collection of instructions, while a process is the actual execution of those instructions. Several processes may be associated with the same program, for example opening up two windows of the same program typically means two processes are being executed. A single computer processor executes only one instruction at a time. To allow users to run several programs at once, single-processor computer systems can perform time-sharing - processes switch between being executed and waiting to be executed. In most cases this is done at a very fast rate, giving an illusion that several processes are executing at once. Using multiple processors achieves actual simultaneous execution of multiple instructions from different processes, but time-sharing is still typically used to allow more processes to run at once. For security reasons most modern operating systems prevent direct inter-process communication, providing mediated and limited functionality. However, a process may split itself into multiple threads that execute in parallel, running different*

*instructions on much of the same resources and data. This is useful when, for example, it is necessary to make it seem that multiple things within the same process are happening at once (such as a spell check being performed in a word processor while the user is typing), or if part of the process needs to wait for something else to happen (such as a web browser waiting for a web page to be retrieved).*

Foldoc (2008): *The sequence of states of an executing program. A process consists of the program code (which may be shared with other processes which are executing the same program), private data, and the state of the processor, particularly the values in its registers. It may have other associated resources such as a process identifier, open files, CPU time limits, shared memory, child processes, and signal handlers. One process may, on some platforms, consist of many threads. A multitasking operating system can run multiple processes concurrently or in parallel, and allows a process to spawn "child" processes.*

**Real time**

WikipediaWS (2008): *In computer science, real-time computing (RTC) is the study of hardware and software systems which are subject to a "real-time constraint"—i.e., operational deadlines from event to system response. By contrast, a non-real-time system is one for which there is no deadline, even if fast response or high performance is desired or even preferred. The needs of real-time software are often addressed in the context of real-time operating systems, and synchronous programming languages, which provide frameworks on which to build real-time application software. A real time system may be one where its application can be considered (within context) to be mission critical.* (…) *Real-time computations can be said to have failed if they are not completed before their deadline, where their deadline is relative to an event. A real-time deadline must be met, regardless of system load.*

**Reliability (survivability)**

*The continuity of correct service* (Avizienis, 2004).

**Scalability (survivability)**

*Systems must be capable of adapting to a range of operations, from local operation to widely dispersed systems, from users to a considerable community of users, from a closed community to an open-system environment. Where single approaches are not applicable, the parameterized configuration should permit adaptation according to the specific requirements* (Neumann, 2000).

**Security (survivability)**

According to Avizienis et al. (2004) *security is a composite of the attributes of confidentiality, integrity, and availability, requiring the concurrent existence of 1) availability for authorized actions only, 2) confidentiality, and 3) integrity with "improper" meaning "unauthorized.".*

Neumann (2000) identifies two types of security: system security and data security. He also identifies some important forms of security: confidentiality, integrity, availability, authentication, thrusted paths, prevention of misuse, tamperproofing and anti-reverse-engineering techniques and auditability and detection of misuse. *Security must encompass dependable protection against all*

*relevant concerns, including confidentiality, integrity, availability despite attempted compromises, preventing denials of services, preventing and detecting misuse, providing timely responses to perceived threats, and reducing the consequences of unforeseen threats. It includes both system security (e.g. protecting systems and networks against tampering and other misuses) and information security (e.g. protecting data and programs against tampering and other misuses)* (Neuman, 2000).

**Service failure (survivability)**

*A service failure, often abbreviated here to failure, is an event that occurs when the delivered service deviates from correct service. A service fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function* (Avizienis, 2004). *A service failure is a transition from correct service to incorrect service. The same authors propose additional definitions:*

- ***Content failures**: The content of the information delivered at the service interface (i.e., the service content)deviates from implementing the system function.*
- ***Consistent failure**: The incorrect service is perceived identically by all system users.*
- ***Development failure**: Development failure causes the development process to be terminated before the system is accepted for use and placed into service. There are two aspects of development failures: 1. Budget failure. The allocated funds are exhausted before the system passes acceptance testing. 2. Schedule failure. The projected delivery schedule slips to a point in the future where the system would be technologically obsolete or functionally inadequate for the user's needs.*
- ***Erratic failure**: When a service is delivered (not halted), but is erratic (e.g., babbling).*
- *Halt failure: Or simply halt, when the service is halted (the external state becomes constant, i.e., system activity, if there is any, is no longer perceptible to the users); a special case of halt is silent failure, or simply silence, when no service at all is delivered at the service interface (e.g., no messages are sent in a distributed system).*
- ***Inconsistent failure**: Some or all system users perceive differently incorrect service (some users may actually perceive correct service); inconsistent failures are usually called, after [38], Byzantine failures.*
- ***Timing failure**: The time of arrival or the duration of the information delivered at the service interface (i.e., the timing of service delivery) deviates from implementing the system function.*

**Service failure mode (survivability)**

*The deviation from correct service may assume different forms that can be called "modes" and ranked according to failure severity* (Avizienis, 2004).

**Survivability (survivability)**

*The concept of survivability is an emergent property. It is the ability of a system to satisfy and continue to satisfy a number of critical requirements in the face of adverse conditions (intentional*

*or not).* (...) *It denotes the ability of a computer-communication system-based application to continue satisfying certain critical requirements (e.g. requirements for security, reliability, real-time responsiveness, and correctness) in face of adverse conditions.* (...) *It applies to national infrastructures, computer-communication infrastructures and to underlying computer systems and communication systems* (Neumann, 2000). This author identifies 5 types of survivability, they are:

- ***Information survivability****: the extent to which suitably correct and up-to-date information can be available whenever it is needed*
- ***Computer-system survivability****: the extent to which a computer system's ability can continue to satisfy certain stated requirements in the face of arbitrary adversities*
- ***Network survivability****: the extent to which a computer network's ability can continue to satisfy certain stated requirements in the face of arbitrary adversities*
- ***Application service survivability****: the extent to which the services provided by an entire system application attain system survivability*
- ***Enterprise survivability****: the extent to which an overall enterprise can continue to satisfy certain stated requirements in the face of arbitrary adversities.*

**sysfs (Linux)**

WikipediaWS (2008): *Sysfs is a virtual file system provided by Linux 2.6. Sysfs exports information about devices and drivers from the kernel device model to userspace, and is also used for configuration.*

**Threat (survivability)**

*The use of threats, for generically referring to faults, errors, and failures has a broader meaning than its common use in security, where it essentially retains it usual notion of potentiality. In our terminology, it has both this potentiality aspect (e.g., faults being not yet active, service failures not having impaired dependability), and a realization aspect (e.g., active fault, error that is present, service failure that occurs)* (Avizienis, 2004).

**Total state of a system**

*The set of the following states: computation, communication, stored information, interconnection, and physical condition* (Avizienis, 2004).

**Trust (survivability)**

*Trust is something you attribute to a system entity, whether that entity is trustworthy or not* (Neumann, 2000). *It can be defined as accepted dependence* (Avizienis, 2004).

**Trusted paths (survivability)**

*In conventional systems, a user has no real assurance that he or she is actually communicating with the desired system (rather than masquerader or Trojan horse system), and one system has little assurance that it is actually communicating with a second system of its choice (rather than a*

*masquerader or accidental alternative). A communication path whose teminations can be assured without compromise is known as a trusted path.* (...) *Trusted paths must be persistent as well as initially authenticated. Other risks include covert channels and timing attacks related to trusted paths.* (Neuman, 2000).

**Trustworthiness (survivability)**

*Trustworthiness is a measure of how extensively a given module, system, network, or other entity deserves to be trusted to satisfy its stated requirements when confronted with arbitrary threats* (Neuman, 2000). *A trustworthy entity deserves to be trusted. For this author, the notion of trustworthiness encompasses all aspects of survivability and its subtended requirements.*

**Vulnerability (survivability)**

*An internal fault that enables an external fault to harm the system* (Avizienis, 2004)

# Distribution list

Document No.: DRDC Valcartier TR 2008-300

**LIST PART 1: Internal Distribution by Centre**

3 Document library
1 Guy Turcotte
1 Jean-Claude St-Jacques
1 Robert Charpentier
1 Mario Couture
1 Frédéric Michaud
1 Frédéric Painchaud
1 Nawel Chefai
1 Philippe Charland
1 Martin Salois

12 TOTAL LIST PART 1

**LIST PART 2: External Distribution by DRDKIM**

1 Library and Archives Canada
1 Dr Julie Lefebvre, DRDC Ottawa, 3701 Carling Avenue, Ottawa, Ontario, K1A 0Z4
1 Chris McMillan, DRDC Ottawa, 305 Rideau Street, Ottawa, Ontario, K1A 0K2
1 Paul Béland, DRDC Ottawa, 305 Rideau Street, Ottawa, Ontario, K1A 0K2
1 LCol. J.M. Drapeau, CFNOC, NDHQ 101 Col By dr. Ottawa K1A 0K2
1 Paul Lamoureux, DRDC Ottawa, NDHQ 101 Col By dr. Ottawa K1A 0K2
1 Dominique Toupin, Ericsson (see 1 below)
1 Dr. Michel Dagenais, Polytechnique de Mtl (see 2 below)
1 Dr. Béchir Ktari, Université Laval (see 3 below)
1 François Chouinard, Ericsson (see 1 below)
1 Mathieu Desnoyers, Polytechnique de Mtl (see 2 below)
1 Gabriel Matni, Polytechnique de Mtl (see 2 below)
1 François Prenoveau, Polytechnique de Mtl (see 2 below)
1 François Lajeuness-Robert, Université Laval (send to B. Ktari) (see 3 below)
1 Dr. Timothy C. Lethbridge, University of Ottawa (see 4 below)
1 Dr Abdelwahab Hamou-Lhadj, Concordia University (see 5 below)
1 Dr. Alex Navarre, NSERC (see 6 below)
1 Dr. Robert Roy, Polytechnique de Mtl (see 2 below)
1 Dr. Marc Khouzam, Ericsson (see 1)
1 Pierre-Marc Fournier, Polytechnique de Mtl (see 2 below)
1 DRDKIM (PDF)

(1) Ericsson, 8400 Decarie Blvd., Town of Mont Royal, QC, H4P 2N2
(2) Polyth. Montréal, Dpt. gen. inf./log., C.P. 6079, succ. Centre-Ville Montréal, QC, H3C 3A7
(3) Laval Univ., 2325 rue de l'Université, Québec, Qc, G1V 0A6
(4) Ottawa Univ., 75 Laurier Ave. E., Ottawa, ON., K1N 6N5
(5) Concordia Univ., Dpt. El. Comp. Eng., 1455 de Maisonneuve Blvd. W., Montreal, Qc, H3G 1M8
(6) NSERC, 350 Albert Street, Ottawa, ON, K1A 1H5

22 TOTAL LIST PART 2

**34 TOTAL COPIES REQUIRED**

This page intentionally left blank.

## DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

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.)

Defence R&D Canada – Valcartier
2459 Pie-XI Blvd North
Quebec (Quebec)
G3J 1X5 Canada

2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.)

UNCLASSIFIED

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.)

Tracing, monitoring and analysis of distributed multi-core systems: Selected feasibility studies

4. AUTHORS (last name, followed by initials – ranks, titles, etc. not to be used)

M. Couture, Prenoveau, F., Lajeunesse-Robert, F.

| 5. DATE OF PUBLICATION (Month and year of publication of document.) | 6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.) | 6b. NO. OF REFS (Total cited in document.) |
|---|---|---|
| April 2009 | 127 | 78 |

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.)

Technical Report

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

Defence R&D Canada – Valcartier
2459 Pie-XI Blvd North
Quebec (Quebec)
G3J 1X5 Canada

| 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.) | 9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.) |
|---|---|
| 15BZ | |

| 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.) | 10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.) |
|---|---|
| DRDC Valcartier TR 2008-300 | |

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

Unlimited

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.))

Unlimited

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.)

Monitoring, tracing and analysis of software execution are indispensable activities for the surveillance, protection and optimization of our national computing infrastructures. However, in recent years, our ability to monitor and analyze software execution has been seriously disrupted by the emergence of multi-core CPUs and the higher level of interconnectivity (between networked systems). These complex systems are already being deployed in our command and control operations. They operate at a much higher transaction rate and they fragment and execute their computing and communication tasks in parallel, leading to huge and extremely complex execution traces to analyze. Current analysis technology is thus overwhelmed by the new computing capability of these systems.

Studies were conducted in 2008 in order to better define the next R&D effort that will be undertaken to address this problem. Among other things, the studies examined the feasibility of developing a feedback-directed diagnostic system based on the *LTTng* framework. Important risks associated with critical technical aspects of this R&D effort were identified and reduced. This document describes the work that was done as well as results and recommendations resulting from these feasibility studies.

Le suivi, le traçage et l'analyse de l'exécution logicielle sont des activités indispensables pour la surveillance, la protection et l'optimisation dans le cadre de notre infrastructure nationale informatique. Néanmoins, ces dernières années, notre capacité à suivre et analyser l'exécution logicielle a été profondément affectée par l'émergence des unités de calcul (CPU) multi cœurs et le haut niveau d'interconnexion (entre les systèmes mis en réseau). Ces systèmes complexes sont déjà déployés dans nos opérations de commandement et contrôle. Leur fonctionnement implique des taux de transaction plus élevés ; ceux-ci fragmentent et exécutent leurs tâches de communication et de calcul en parallèle, ce qui résulte en d'énormes traces d'exécution complexes à analyser. La technologie d'analyse actuelle est donc dépassée par la nouvelle capacité de calcul de ces systèmes.

Des études ont été réalisées en 2008 afin de mieux définir les prochains efforts de R et D qui seront déployés dans le but d'aborder ce problème. Parmi celles-ci, la possibilité de développer un système de diagnostic basé sur la rétroaction et utilisant l'environnement *LTTng* a été étudiée. Les risques importants associés aux aspects critiques de cet effort de R et D ont été identifiés et réduits. Ce document énumère et décrit le travail réalisé ainsi que les résultats et recommandations résultant de ces études de faisabilité.

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.)

Monitoring, tracing, analysis, formalism, execution trace, multi-core, distributed, information system