On Monitoring Asynchronous Components, Asynchronously

ANONYMOUS AUTHOR(S)

Runtime Monitoring is becoming an important analysis tool for improving software quality. The prevailing opinion within the software development community is that inline monitoring is preferred over outline monitoring, mainly because it leads to lower runtime overheads. This paper argues that software has evolved enough over the last few years to put this commonly-held view into question. We provide a series of qualitative arguments in favour of outline monitoring in the case of component-based distributed software. We also develop an algorithm for the correct outline monitoring of dynamic decentralised systems. Finally we conduct a rigorous analysis of the overheads induced by both inline and outline monitoring over models of component-based systems, which enables us to assess more precisely the overhead discrepancy induced by the two variants of the runtime analysis technique.

Additional Key Words and Phrases: Asynchronous component systems, Decentralised monitoring, Dynamic reconfiguration

1 INTRODUCTION

Software has changed dramatically over the last decades. The rise of the app economy on mobile devices, the widespread use of streaming services, together with the impending wave of IoT, have fundamentally altered the manner in which software is developed, the tasks it is expected to conduct, and the environments in which it is required to execute. In these cases, software runs autonomously, under constrained resources, and in decentralised fashion. Viewed globally, this software is structured as a collection of encapsulated components that are massively replicated [Josuttis 2007]; they are expected to run without interruptions for days, months or even years, and scale in response to fluctuating circumstances [Garg 2015]. These components interact with one another via asynchronous messaging [Hohpe and Woolf 2003] (*e.g.* as microservices [Jamshidi et al. 2018]). Invariably, these components are developed by third parties using different technologies.

This landscape poses new challenges to developers. Software is expected to adhere to stringent requirements (*e.g.* streaming services need to ensure adequate levels of QoS) and increasingly handles sensitive information (*e.g.* mobile devices access our financial data, and medical implants regulate insulin levels), raising the stakes of understanding how it really behaves at runtime. At the same time, the behaviour of component-based software has become harder to understand and predict. This is due to a number of reasons. First, the proper functioning of a system does not depend solely on that of its individual components, but also relies on the manner in which they are *integrated* with one another; this information is rarely readily available when components are provided by third parties (*e.g.* webservices or binary libraries), or when their connections are determined at runtime (*e.g.* dynamic service discovery). Second, the sheer scale and distribution of said software further complicates the acquisition and comprehension of this information. Third, these systems execute in open environments, where they are subject to malicious attacks from adversaries that are hard to model and anticipate statically.

Traditional verification approaches like model checking—conceived for software developed in monolithic fashion with conventional deployment practices—do not apply (at least, not in their present form). They are also bound to suffer from the usual scalability issues for large code-bases. Popular methods such as testing and mocking are also largely ineffective when debugging open, large-scale distributed systems with a multitude of execution paths [Alshahwan et al. 2019; Alvaro et al. 2016; Arora et al. 2016]. Rather, these factors have increased the need to complement the

---

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/

verification analysis carried out at the design and development phases with validation at the *post-deployment* phase. For instance, techniques that traditionally scale well, such as type-systems, have evolved to support the integration of dynamic analyses [Siek and Taha 2007; Takikawa et al. 2012]. Unfortunately, these technologies are inherently language-specific and, presently, are not mature enough to cope with software developed using multiple programming languages.

Interestingly, industry has witnessed a profusion of tools that enable the *observation and monitoring* of such systems *at runtime*. These technologies are broadly classified under Application Performance Monitoring tools (APMs) [Heger et al. 2017]. They include commercial solutions such as DataDog, Instana and New Relic One, platform-specific frameworks such as inspectIT-Ocelot (a JVM Agent) and WombatOAM (for Erlang/OTP), and open source offerings such as Dapper, and Zipkin. These tools extend traditional profilers to support distributed tracing and telemetry, log aggregation, data storage, processing and presentation, anomaly detection and threshold-violation alerting, root cause isolation, and also automation for runtime system adaptation. APMs are used extensively for maintenance and performance tuning to identify hotspots and reduce bottlenecks; they presently have an edge on static analysis tools for critical-path analysis and unearthing performance anti-patterns [Smith and Williams 2001, 2002]. Reported load-time errors and statistics on end-to-end response times are used to improve user experience. The tracing of events such as exceptions and process failures is used for debugging (live or offline), whereas audit trails are used for forensic analysis in the case of security breaches. APMs may also turn program information that used to be ephemeral and uncertain into something that is concrete and analysable through Machine Learning technologies.

The verification counterpart to APMs is Runtime Verification (RV) [Bartocci et al. 2018], where executable code is synthesised from formal specifications to observe the behaviour of a running system against said specifications. Although there is a clear case for using RV for decentralised and distributed scenarios [Francalanza et al. 2018; Sánchez et al. 2019] there is one fundamental difference between present-day distributed RV approaches and APMs. Concretely, most APMs operate as *external entities*, running *asynchronously* to the system under scrutiny (SuS) to analyse its behaviour via intermediaries such as log files and data warehouses. By treating the SuS as a black-box, APMs become largely programming-language agnostic. Moreover, by operating externally, APMs provide added assurances that their monitoring does not directly interfere with the execution of the SuS. In contrast, the state-of-the-art in decentralised and distributed RV [Colombo et al. 2009; El-Hokayem and Falcone 2020; Jin et al. 2012; Kim et al. 2001; Reger et al. 2015; Sen et al. 2004, 2006] is dominated by tools that still runs synchronously to the SuS, typically using weaving via code injection (inlining). One reason for this is that most efforts are extensions of mature tools that were originally developed for local, single-threaded RV. There, inlining is the preferred method of instrumentation [Bartocci et al. 2018] because it yields lower overheads [Cardoso et al. 2017, 2016]; seminal work in security also highlights the advantages inline instrumentation begets when analysing insecure software [Erlingsson 2004; Erlingsson and Schneider 1999]. However, inlining and synchronous instrumentation may not necessarily be the best approach for decentralised and distributed monitoring. For instance, inlining relies on assumptions, such as full access to the SuS source code, that may not always be possible in this setting; inlining is also programming-language dependent and difficult to administer on heterogenous distributed systems; it is also more intrusive and harder to undo once a properly running SuS is attained.

Despite the fact that low overheads are a central concern for any monitoring system, this paper contends that the prevailing view about inline and outline monitoring warrants revisiting. To this end, we present a detailed study of asynchronous monitoring, where instead of considering the analysis aspect of the problem (see [Francalanza et al. 2018] for a detailed survey), we focus on the instrumentation part that determines how the runtime analysis hooks onto the running system. To our knowledge, this aspect of RV has seldom been studied in its own right, even though it probably contributes more to runtime overheads than the runtime analysis itself. Concretely, we make the following contributions:

(1) We detail an algorithm for concurrent asynchronous monitoring that scales in line with a SuS that grows and shrinks. We make minimal assumptions on the operational model to ensures that our algorithm is sufficiently general to be instantiated in a variety of languages and technologies; the algorithm is also agnostic of the runtime analysis carried out, making it applicable for monitoring both functional and non-functional requirements alike (Sec. 3).
(2) We build models to evaluate outline monitoring *quantitatively*. We use a series of systematic experiments that compare it with inline monitoring, for a selection of typical system loads. Although we confirm that inline monitoring induces lower overhead, we debunk the generally-held assumption that asynchronous monitoring is necessarily infeasible. We are unaware of similar results within the scientific community (Sec. 4).

2 BACKGROUND

Online monitors analyse the execution of the SuS while it is running. The analysis is typically moves forwards in time, discarding already processed events of interest to keep monitors as lightweight as possible. Depending on the monitoring application, trace events can be analysed within the system itself at the point they occur (inline monitors), or transmitted to an external entity that performs the analysis without the system (outline monitors). Inline monitors necessarily execute in synchronous fashion with the SuS. Outline monitors typically execute asynchronously in a separate thread, but can also run synchronously when sharing a common thread with the SuS, or in lock-step via a handshake communication protocol.

The nature of overheads. Inlining determines *statically* the points in the system where the events of interest occur, and the monitor instructions are injected accordingly; these segments lay dormant and get activated only when certain execution paths are followed. Outline instrumentation defers this decision until runtime. In order to scale dynamically, it needs to analyse every event generated by the SuS to determine whether to instrument additional monitors; this is clearly more flexible but also more expensive.

Intrusiveness. Code injection via inlining relies on elements of the program structure. In Object Orientated (OO) programs, where the unit of decomposition is the object, code weaving patterns like Aspect-Oriented Programming (AOP) [Kiczales et al. 1997] package *aspect* code into objects that interact with other system objects through method invocation. In concurrent paradigms, systems are structured as *independent* process units such as actors that interact either via asynchronous message passing or via synchronous mechanisms (*e.g.* channels or locks). In either case, the interaction between processing entities is determined *at runtime* by the scheduler. This complicates the task of inlining monitor code since this code has to account for these interactions; this becomes even harder to manage when inlined monitors themselves interact as well. Outlining naturally keeps monitors and system processes separate, reducing the risks of subtle bugs from occurring when runtime monitors are introduced.

Separation of Concerns. The separation of monitors and system processes as distinct computational units (induced by outlining) adheres better to established software engineering principles. Inlined monitors are sometimes perceived as functionality that can be aspectised in order to organise the system and monitor code at the software design level. This separation however, does not permeate down to the runtime level, since both system and monitor code executes on the *same* thread. A dependency is created between the two, such that if a monitor embedded in one system process crashes, so does the process; the reverse is also possible, and the runtime analysis is lost. Moreover, inlined monitors are harder to remove or disable in a running system once weaved.

Flexibility. There are cases where monitors cannot be inlined because code injection is not possible. In a setting such as ours, certain components might be offered as-a-service or in the form of a commercial library where code modifications are prohibited due to availability or licensing agreements. Obfuscated third-party libraries, while possible to reverse engineer [Chen and Chen 2006], are be hard to instrument, and this certainly cannot be accomplished without intimate knowledge of the decompiled binary instructions. In instances where it can be done, monitor inlining generally demands a *redeployment* of the instrumented system components, which could be infeasible for long-running systems. Outline monitoring often relies on *tracing* as a mechanism to acquire runtime information about the SuS. One advantage that many tracing frameworks offer is the capability of dynamically switching tracing on or off *without* the need to recompile or redeploy the traced system. Tracing is typically intended for use in production due to the minimal levels of overhead it induces; this also makes it an invaluable tool when it comes to detect and analyse problems that occur in real-time. Many programming language frameworks come equipped with tracing mechanisms that can be configured programmatically (*e.g.* Erlang). There are also tracing frameworks such as and LTTng [Desnoyers and Dagenais 2006] and DTrace [Cantrill 2006; Cantrill et al. 2004] that work at the operating system level; DTrace for instance, also supports tracing at the application (*e.g.* MySQL, Firefox) and programming language levels (*e.g.* C, Java, Erlang, *etc.*).

3 DECENTRALISED OUTLINE MONITORING

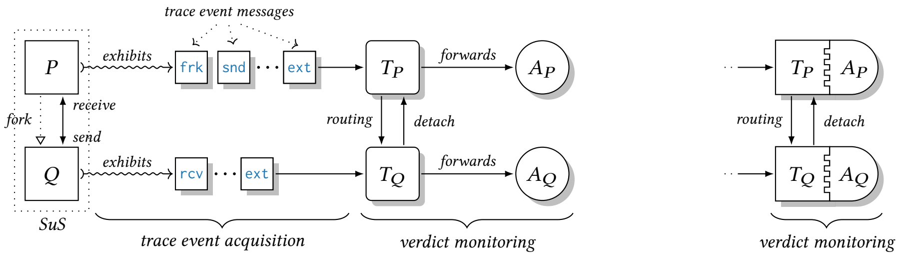
We present an outline monitoring algorithm to analyse the behaviour of the SuS by observing its components in a *decentralised* fashion. Our solution rests on these general assumptions:

- A_1 *No global clock.* System components are not synchronised through a common clock.
- A_2 *System is dynamic.* The number of system components may fluctuate at runtime.
- A_3 *Messages can be reordered.* This does *not* apply for point-to-point communication: successive messages between the same source and destination are delivered in the sequence issued.
- A_4 *Communication is reliable.* Messages sent are not tampered with, and communication links never fail (*i.e.,* message delivery is guaranteed and messages duplication does not arise).
- A_5 *Components are reliable.* Components never fail-stop or exhibit Byzantine failures.

Our investigation is scoped to execution-monitors/sequence recognisers [Ligatti et al. 2005; Schneider 2000] where monitors reach irrevocable verdicts after observing a *finite* sequence of system trace events [Aceto et al. 2019b]. We want our monitors to abide by the following requirements:

- R_1 *Monitoring is passive* and only reacts to SuS events.
- R_2 *Monitoring should minimise interference on SuS execution.*
- R_3 *Monitoring is decentralised* without a central coordinating entity.
- R_4 *Monitoring does not miss events or analyse them out of order.*

Monitors are instrumented to run in *asynchronous* fashion, in line with assumption A_1 and what is normally found in distributed setups; although this is outside the scope of our present investigation, distribution could be obtained by weakening assumptions A_4 and A_5. Asynchrony may occasionally affect timely detections. Assumption A_2 and requirements R_2 and R_3 also call for monitoring to *scale dynamically*, continually reconfiguring its choreography in response to certain events exhibited by the SuS whilst the runtime analysis is in progress. This complicates outline monitoring substantially, since it must contend with the potential race conditions that may arise. Requirement R_4 addresses



(a) Tracer and analyser organised into separate processes (b) Tracer and analyser as a single process

Fig. 1. Outline verdict monitoring set-up consisting of tracer and analyser roles

problems caused by assumption A_3. It is vital for execution-monitors, which are usually sensitive to the temporal ordering of the observed events (*e.g.* RV, root cause analysis, *etc.*).

3.1 Overview

We proposed *two* outline monitoring set-ups. The choreography in fig. 1a, consisting of independent *tracer* and corresponding *analyser* processes, teases apart the task of trace event routing and monitor reorganisation, performed by tracers, from the task of trace event examination, effected by the analysers. This separation of concerns favours the single responsibility [Agha et al. 1997; Martin 2013] design approach at the expense of introducing an extra process into the monitoring set-up. By contrast, fig. 1b merges the tracing and analysis tasks to forgo this extra process. Our outline approach assumes the existence of a *tracing mechanism* that provides streams of execution events in the form of *messages* for the components of interest in the SuS. The mechanism also allows tracers to control the tracing configuration dynamically at runtime (see discussion in sec. 2). In fig. 1, trace event messages are shown as issuing from processes P and Q and directed to their respective tracers T_P and T_Q; these messages are forwarded to monitors A_P and A_Q for analysis in fig. 1a, or analysed directly as in fig. 1b. The tracing portion of our algorithm relies on these assumptions:

A_6 *Tracers cannot share system processes.* A system process can be traced by (*i.e.*, trace event messages are sent to) *at most one* tracer at any point in time.

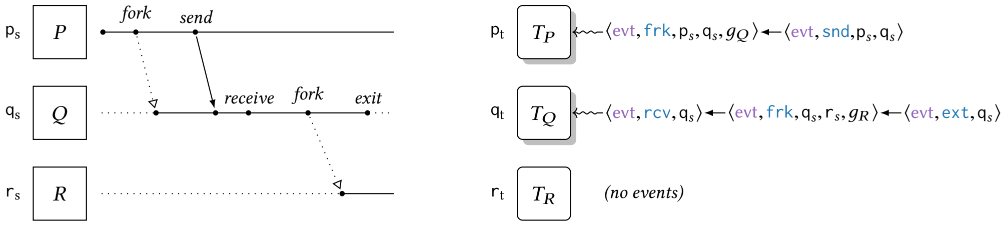
A_7 *System processes may share tracers.* A tracer may trace more than one system process.

A_8 *System processes inherit tracers.* A system process that is forked by another process that is being traced becomes automatically traced by the *same* tracer.

Assumption A_6 means that for one tracer to start tracing a system process currently being traced, it must first *stop* the active tracer before it can take over and continue tracing this process itself.

3.2 Definitions and Notation

Processes. We assume a denumerable set of process identifiers (PIDs) to uniquely refer to processes. We distinguish between system, tracer and analyser process forms, denoting them respectively by the sets $\textsc{Pid}_s$, $\textsc{Pid}_t$ and $\textsc{Pid}_a$, where $p_s \in \textsc{Pid}_s$, $p_t \in \textsc{Pid}_t$, $p_a \in \textsc{Pid}_a$. New processes are created via the function fork(g) that takes the signature of the code to be run by the forked process, $g \in \textsc{Sig}$, returning its *fresh* PID. We refer to the process invoking fork as the *parent*, and to the forked process as the *child*. To create monitor processes, the function fork is overloaded to accept verdict-flagging code, $v \in \textsc{Mon}$, and return the corresponding PID p_a; tracer processes are spawned analogously. Processes communicate with one another through asynchronous messages. Each process is equipped with



(a) Process creation sequence for P, Q and R

(b) Trace event streams for T_P, T_Q and T_R (see tbl. 4)

Fig. 2. SuS with processes P, Q, and R instrumented with three independent monitors

a *message queue*, K, from where it can read messages *out-of-order* and in a *non-blocking* fashion. Unless stated otherwise, we use the terms *tracer* and *analyser* synonymously since the distinction between the two notions is unimportant for the remainder of this section.

Messages. Messages, $m \in \textsc{Msg}$, are represented as tuples $\langle q, d_0, d_1, \dots, d_n \rangle$, where q is a *message qualifier* indicating the message type, and $d_{i \in \mathbb{N}}$ are the data elements comprising the message payload. We classify between three messages types, $q \in \{\text{evt}, \text{dtc}, \text{rtd}\}$, described thus:

$q = \text{evt}$: *trace events* obtained via the tracing mechanism to be analysed;
$q = \text{dtc}$: *detach commands* that tracers exchange to reorganise the monitoring choreography;
$q = \text{rtd}$: trace event or command messages that are *routed* between tracers.

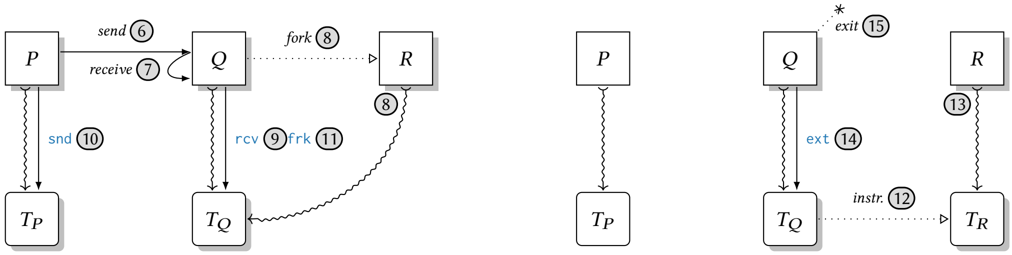
The meta-variables e, c, and r are reserved to refer to messages of types evt, dtc and rtd respectively We use the suggestive dot notation (.) to access specific data elements through indexable *field names* (*e.g.* the message qualifier is accessible through m.type). Trace event messages are structured as $\langle q{=}\text{evt}, d_0{=}a, d_1, \dots, d_n \rangle$, where $a \in \textsc{Act}$ identifies the kind of action exhibited by the SuS, and $d_1, \dots, d_n$ designate the data particular to the event. For our exposition, we let $\textsc{Act} = \{\texttt{frk}, \texttt{ext}, \texttt{snd}, \texttt{rcv}\}$, respectively denoting the process actions fork (`frk`), exit (`ext`), send denoted via "!" (`snd`) and receive (`rcv`). We abuse the notation and use a in lieu of the full trace event message data (*i.e., q* and $d_1, \dots$) when this simplifies the explanation. The data elements particular to the four trace events are accessed using the field names catalogued in tbl. 4 of app. A.

3.3 The Monitoring Approach

We present our outline decentralised monitoring algorithm incrementally, highlighting the issues that arise when the monitoring choreography reorganises itself as the SuS executes. The algorithm covers *both* arrangements outlined in fig. 1. In the pseudocode, we also highlight the technical differences between the two variants, namely: *(i)* whether trace events are analysed by the separate analyser, as in fig. 1a, or directly by the tracer, as in fig. 1b, and; *(ii)* depending on the variant, whether or not the separate analyser is created and terminated. The core logic found in each monitor in a choreography is described in lsts. 1–3; auxiliary logic may be found in app. A. Our exposition focusses on the tracer logic, and is agnostic of the analyser code. Each tracer state comprises of three maps: the *routing map*, Π, describing how to re-route events to other tracers, the *instrumentation map*, Φ, describing when new monitors need to be launched, and a *tracked-processes map*, Γ, recording the system processes the tracer is currently monitoring. We detail how these maps are used below.

Dynamic process creation. To reorganise the monitoring choreography as the SuS executes, tracers are programmed to react to specific events observed in the trace; in our setting, these are fork (`frk`)

(a) Process P forks Q; T_P also traces Q (b) T_P instruments new tracer T_Q for process Q



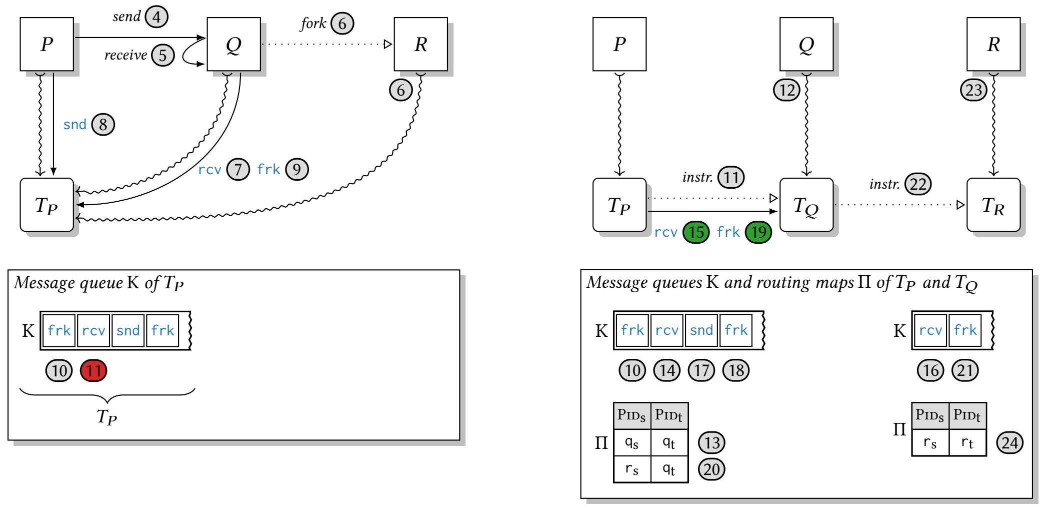
(c) T_P and T_Q analyse trace events independently (d) Processes P, Q, R and corresponding tracers

Fig. 3. Outline tracer instrumentation for processes P, Q and Q (analysers omitted)

and exit (`ext`). System processes are typically created in a *hierarchical fashion*, starting from the top-level level process that forks one or more child processes [Armstrong 2007]; we borrow the standard terminology used to describe the relationships between nodes in a tree (*i.e.,* root, ancestor, descendant, *etc.*) when referring to processes. Fig. 2a depicts our running example where the root P forks a child Q and communicates with it; independently Q spawns R and exits. Our example assumes that a dedicated monitor will be assigned per process; our exposition will focus on the tracers, *i.e.,* T_P, T_Q and T_R in this case, where fig. 2b depicts the order of trace events each of these monitors is expected to analyse.

Trace event acquisition. The tracing mechanism alluded to in sec. 3.1 is defined by the operations Trace, Clear and Preempt listed in lst. 4 of app. A. Trace enables a tracer p_t to register its interest in being notified about trace events of a system process p_s. This operation can be undone using Clear, which blocks the caller, and returns only when all the trace event messages for p_s that are in the process of being delivered are deposited into the message queue of p_t. Preempt combines Clear and Trace, enabling a tracer p_t to take over the tracing of process p_s from another tracer p'_t. Following assumption A_8, tracing is *inherited* by every child process that a traced system process forks; Clear or Preempt can therefore be used to alter this arrangement.

Decentralised trace processing. Fig. 3 demonstrates how the process creation sequence of the SuS can be exploited to systematically instrument tracers and evolve the choreography at runtime. The system processes P, Q and R in fig. 3 are created (with PIDs $\mathsf{p_s}$, $\mathsf{q_s}$, and $\mathsf{r_s}$), following the interaction protocol of fig. 2a. A tracer instruments other tracers whenever it encounters fork events in the execution. In fig. 3a, the root tracer T_P analyses the top-level process P, step ①, and instruments a new tracer, T_Q, for process Q when it observes the fork event $\langle \mathsf{evt}, \mathtt{frk}, \mathsf{p_s}, \mathsf{q_s}, g_Q \rangle$ exhibited by P in step ③. The field e.tgt carried by the fork trace event designates the SuS process that is to be instrumented with the new tracer. Thereafter, T_Q takes over the tracing of process Q by calling



(a) Trace events for P, Q, and R observed by T_P

(b) Trace events for Q routed from T_P to T_Q

Fig. 4. Hop-by-hop trace event routing using local tracer routing maps (analysers omitted)

Preempt with T_P and e.src to continue tracing Q *independently* of T_P, steps ④ and ⑤ in fig. 3b. The root tracer resumes its own analysis in parallel, where it receives the send event $\langle \mathsf{evt}, \mathsf{snd}, \mathsf{p}_\mathsf{s}, \mathsf{q}_\mathsf{s} \rangle$ in step ⑩ after P issues a message to Q in step ⑥. Subsequent fork events observed by T_P and T_Q are handled in the same manner. Figs. 3c and 3d show how the next tracer, T_R, is instrumented as Q forks its child process R. Recall that prior to the instrumentation of tracers T_Q and T_R, processes Q and R automatically start sharing tracers with their respective parents P and Q when forked, as indicated in steps ② and ⑧.

Trace event routing. Different interleaved executions may still arise for the creation sequence depicted in fig. 2a, due to the asynchrony between the SuS and tracer components. Fig. 4 shows an interleaving alternative to the one captured in figs. 3b–3d. In fig. 4a, the root tracer T_P is slow to handle the fork event exhibited by process Q (step ① in fig. 3a), failing to instrument T_Q promptly. Consequently, in fig. 4a, the trace events due to Q are received by T_P in the sequence indicated by steps ⑦ and ⑨. As a result, the receive event $\langle \mathsf{evt}, \mathsf{rcv}, \mathsf{q}_\mathsf{s} \rangle$ is processed by T_P in step ⑪, rather than by the correct tracer T_Q that is eventually instrumented by T_P. This behaviour could derange the runtime analysis, since the events that are expected to be processed by particular analysers unintentionally reach a different monitor.

We address this problem by programming tracers to *filter* the events that are to be analysed locally, and *forward* the rest to other tracers. Fig. 4b shows how the root tracer T_P first instruments T_Q with Q in step ⑪. It subsequently processes the events $\langle \mathsf{evt}, \mathsf{rcv}, \mathsf{q}_\mathsf{s} \rangle$ and $\langle \mathsf{evt}, \mathsf{frk}, \mathsf{q}_\mathsf{s}, \mathsf{r}_\mathsf{s}, g_R \rangle$ in steps ⑭ and ⑱, forwarding them to T_Q, steps ⑮ and ⑲. T_Q acts on these events in steps ⑯ and ㉑, where a second tracer, T_R, is instrumented with R. Concurrently, the event $\langle \mathsf{evt}, \mathsf{snd}, \mathsf{p}_\mathsf{s}, \mathsf{q}_\mathsf{s} \rangle$ is *processed locally* by T_P in step ⑰. Trace event routing is accomplished by maintaining a partial map inside tracers, $\Pi : \textsc{Pid}_\mathrm{s} \rightharpoonup \textsc{Pid}_\mathrm{t}$, relating system and tracer PIDs. A tracer queries its instance of the *routing map* Π for every trace event it processes, to determine whether the event should be handled locally or directed elsewhere. The source PID of the event (field e.src in tbl. 4 of app. A) is used to this effect. Trace events are *forwarded* to the tracer with PID p_t when $\Pi(e.\mathrm{src}) = p_\mathrm{t}$, and

```
1  def Loop$_\circ$($\sigma$, $p_a$)
2    forever do
       # Read routed messages or direct trace events
3      $m \leftarrow$ next message from queue K
4      if $m$.type = evt then
5        $\sigma \leftarrow$ HandleEvent$_\circ$($\sigma$, $m$, $p_a$)
6      else if $m$.type = dtc then
         # dtc command received from descendant
         # tracer: route back to sender
7        $\sigma \leftarrow$ RouteDtc($\sigma$, $m$)
8      else if $m$.type = rtd then
9        $\sigma \leftarrow$ RelayRtd$_\circ$($\sigma$, $m$, $p_a$)
10     end if
11   end forever
12 end def

13 def HandleEvent$_\circ$($\sigma$, $e$, $p_a$)
14   if $e$.act = frk then
15     $\sigma \leftarrow$ HandleFork$_\circ$($\sigma$, $e$, $p_a$)
16   else if $e$.act = ext then
17     $\sigma \leftarrow$ HandleExit$_\circ$($\sigma$, $e$, $p_a$)
18   else if $e$.act $\in$ {snd, rcv} then
19     HandleComm$_\circ$($\sigma$, $e$, $p_a$)
20   end if
21   return $\sigma$
22 end def

23 def HandleFork$_\circ$($\sigma$, $e$, $p_a$)
24   if ($p_t \leftarrow \sigma.\Pi(e.\mathrm{src})) \neq \bot$ then
25     Route($e$, $p_t$)
       # New route for events of child process e.tgt
       # goes through the same tracer p_s of its parent
26     $\sigma.\Pi \leftarrow \sigma.\Pi \cup \{\langle e.\mathrm{tgt}, p_t \rangle\}$
27   else
28     Monitor $p_a$ analyses event $e$
29     $\sigma \leftarrow$ Instrument$_\circ$($\sigma$, $e$, self())
30   end if
31   return $\sigma$
32 end def

33 def HandleExit$_\circ$($\sigma$, $e$, $p_a$)
34   if ($p_t \leftarrow \sigma.\Pi(e.\mathrm{src})) \neq \bot$ then
35     Route($e$, $p_t$)
36   else
37     Monitor $p_a$ analyses event $e$
       # Remove terminated process e.src from group
38     $\sigma.\Gamma \leftarrow \sigma.\Gamma \setminus \{\langle e.\mathrm{src}, \circ \rangle\}$
39     TryGC($\sigma$, $p_a$)
40   end if
41   return $\sigma$
42 end def

43 def HandleComm$_\circ$($\sigma$, $e$, $p_a$)
44   if ($p_t \leftarrow \sigma.\Pi(e.\mathrm{src})) \neq \bot$ then
45     Route($e$, $p_t$)
46   else
47     Monitor $p_a$ analyses event $e$
48   end if
49 end def

Expect: $\sigma.\Pi(c.\mathrm{tgt}) \neq \bot$
50 def RouteDtc($\sigma$, $c$, $p_a$)
51   if ($p_t \leftarrow \sigma.\Pi(c.\mathrm{tgt})) \neq \bot$ then
52     Route($c$, $p_t$)
53     $\sigma.\Pi \leftarrow \sigma.\Pi \setminus \{\langle c.\mathrm{tgt}, p_t \rangle\}$ # Remove route
54     TryGC($\sigma$, $p_a$)
55   end if
56   return $\sigma$
57 end def

58 def RelayRtd$_\circ$($\sigma$, $r$, $p_a$)
59   $m \leftarrow r$.emb
60   if $m$.type = dtc then
61     $\sigma \leftarrow$ RelayDtc($\sigma$, $r$, $p_a$)
62   else if $m$.type = evt then
63     $\sigma \leftarrow$ RelayEvt($\sigma$, $r$)
64   end if
65   return $\sigma$
66 end def

67 def RelayDtc($\sigma$, $r$, $p_a$)
68   $c \leftarrow r$.emb
69   if ($p_t \leftarrow \sigma.\Pi(c.\mathrm{tgt})) \neq \bot$ then
70     Relay($r$, $p_t$)
71     $\sigma.\Pi \leftarrow \sigma.\Pi \setminus \{\langle c.\mathrm{tgt}, p_t \rangle\}$ # Remove route
72     TryGC($\sigma$, $p_a$)
73   end if
74   return $\sigma$
75 end def

Expect: $\sigma.\Pi(r.\mathrm{emb}.\mathrm{src}) \neq \bot$
76 def RelayEvt($\sigma$, $r$)
77   $e \leftarrow r$.emb
78   if ($p_t \leftarrow \sigma.\Pi(e.\mathrm{src})) \neq \bot$ then
79     Relay($r$, $p_a$)
       # New route for events of child process e.tgt
       # goes through the same tracer p_s of its parent
80     if $e$.act = frk then
81       $\sigma.\Pi \leftarrow \sigma.\Pi \cup \{\langle e.\mathrm{tgt}, p_t \rangle\}$
82     end if
83   end if
84   return $\sigma$
85 end def
```

Lst. 1. Tracer loop that handles direct events, message routing and relaying

handled by the tracer itself when no such route exists, *i.e.*, $\Pi(e.\mathrm{src}) = \bot$. HandleFork, HandleExit and HandleComm in lst. 1 implement this logic on lines 24, 34 and 44.

A tracer extends its routing map Π whenever it processes a fork event $\langle \mathsf{evt}, \mathsf{frk}, p_s, p_s', g \rangle$. It has to consider the following two cases:

```
Expect: $e$.act = frk
1   def INSTRUMENT$_\circ(\sigma, e, p_t)$
2     $p_s \leftarrow e$.tgt
3     if $(v \leftarrow \sigma.\Phi(e.\text{sig})) \neq \bot$ then
4       $p_t' \leftarrow$ fork(TRACER$(\sigma, v, p_s, p_t)$)
5       $\sigma.\Pi \leftarrow \sigma.\Pi \cup \{\langle p_s, p_t' \rangle\}$
6     else
        # In $\circ$ mode, there is no process $p_s$ to detach
        # from an ancestor tracer; add $p_s$ to group
7       $\sigma.\Gamma \leftarrow \sigma.\Gamma \cup \{\langle p_s, \circ \rangle\}$
8     end if
9     return $\sigma$
10  end def
```

```
Expect: $e$.act = frk
11  def INSTRUMENT$_\bullet(\sigma, e, p_t)$
12    $p_s \leftarrow e$.tgt
13    if $(v \leftarrow \sigma.\Phi(e.\text{sig})) \neq \bot$ then
14      $p_t' \leftarrow$ fork(TRACER$(\sigma, v, p_s, p_t)$)
15      $\sigma.\Pi \leftarrow \sigma.\Pi \cup \{\langle p_s, p_t' \rangle\}$
16    else
        # Take over $p_s$ from tracer $p_t$; add $p_s$ to group
17      DETACH$(p_s, p_t)$
18      $\sigma.\Gamma \leftarrow \sigma.\Gamma \cup \{\langle p_s, \bullet \rangle\}$
19    end if
20    return $\sigma$
21  end def
```

Lst. 2. Instrumentation operations for direct (○) and priority (●) tracer modes

- C_1 $\Pi(p_s) = \bot$. This means that the tracer needs to adapt the choreography in response to the newly forked process itself. It launches a new (child) tracer $T_{P'}$ with fresh PID p_t' to be instrumented with the forked process p_s', and extends Π with the mapping $p_s' \mapsto p_t'$; or,
- C_2 $\Pi(p_s) = p_t'$. This means that a route to the tracer with PID p_t' exists for events originating from p_s. Accordingly, the tracer forwards the fork event for p_s to p_t', and again extends Π with the mapping $p_s' \mapsto p_t'$ (*i.e.,* future events from the new process p_s' will also be forwarded to p_t').

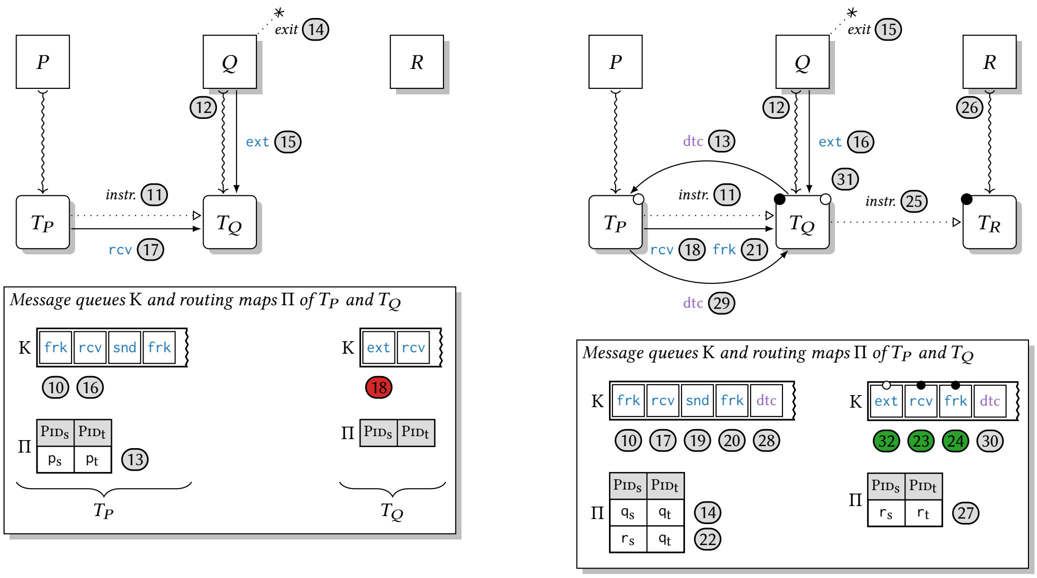
Fig. 4b depicts the routing maps of tracers T_P and T_Q. T_P adds the mapping $\mathsf{q}_s \mapsto \mathsf{q}_t$ in step ⑬, after handling the event $\langle$evt, frk, $\mathsf{p}_s, \mathsf{q}_s, g_Q \rangle$ to instrument T_Q with Q in steps ⑩ and ⑪; this is an instance of case C_1. Lst. 2 describes function INSTRUMENT where, on line 5, the mapping $e.\text{tgt} \mapsto p_t'$ (with $e.\text{tgt} = p_s'$) is added to Π following the creation of tracer p_t'. Step ⑳ of fig. 4b constitutes an instance of case C_2. T_P adds the map $\mathsf{r}_s \mapsto \mathsf{q}_t$ after processing $\langle$evt, frk, $\mathsf{q}_s, \mathsf{r}_s, g_R \rangle$ for R, step ⑱. Crucially, T_P *does not instrument* a new tracer, but simply delegates this task to T_Q by forwarding the event in question. Lines 26 and 81 in lst. 1 (and later line 26 in lst. 3) are manifestations of this, where the mapping $e.\text{tgt} \mapsto p_t'$ is added after the fork event e is routed to the next tracer p_t'.

We note that in fig. 4b both mappings inside T_P, created in steps ⑬ and ⑳, point to tracer T_Q, and the mapping ㉔ in T_Q points to T_R. This routing map configuration arises as a result of cases C_1 and C_2, and implies that any given tracer can *only* forward trace events to adjacent neighbours. For instance, trace events exhibited by R (to be collected by T_P) need to be forwarded twice to reach the intended tracer T_R: from tracer T_P to T_Q, and from T_Q to T_R. This *hop-by-hop routing* [Baker 1995] between tracers forms a connected DAG, and ensures that every message is *eventually delivered* by the tracer choreography. Our algorithm performs routing using two operations, ROUTE and RELAY from lst. 5 in app. A. ROUTE creates a new message, r, with type rtd and embeds the message that needs to be routed. Messages routed to a tracer can *either* be analysed or forwarded using RELAY.

Trace event order preservation. Trace event routing does not guarantee that a tracer will receive events in the sequence that should be processed by each monitor, as depicted in fig. 2b, in order to reflect the execution of the SuS shown in fig. 2a. The situation arises when the tracer *simultaneously* actively traces a system component while receiving routed events for that component from another tracer. Fig. 5a highlights the deleterious effect this can have on the runtime analysis should events be deposited out-of-order in the tracer's message queue (assumption A_3). Tracer T_Q takes over tracing process Q from T_P in step ⑫, and collects the event ext, step ⑮, *before* it receives the routed event rcv for Q in step ⑰. If T_Q naïvely analyses the events based on their position in the message queue, step ⑱, it would violate the (correct) order stated in fig. 2b; in fact Q *cannot* receive a message after exiting. To address this issue, tracers *prioritise* the processing of routed trace event messages. This captures the invariant that out of all events to be analysed by a monitor, routed events must have temporally preceded all other events. Tracers operate on two levels, *priority* mode and *direct* mode, denoted by • and ∘ in our algorithm. Fig. 5b shows that when in priority mode, tracer T_Q dequeues the routed events `rcv` and `frk` (labelled with •), and handles them first: `rcv` is analysed in step ㉓, whereas `frk` results in the instrumentation of a new tracer T_R in step ㉕. Events that should not be analysed by the tracer are forwarded as described earlier in fig. 4. We note that T_Q can still receive trace events from process Q while this is ongoing, but these events are only considered once the tracer transitions to direct mode later. Newly-instrumented tracers default to *priority* mode, so that routed trace events are processed first (see line 5 in lst. 6 of app. A).

Lst. 3 shows the priority processing logic, Loop$_\bullet$, where routed trace events are dequeued and handled (lines 3 and 6). HandleFork, HandleExit and HandleComm for the two tracer modes, Loop$_\circ$ and Loop$_\bullet$ in lsts. 1 and 3, handle trace events differently. In priority mode, tracers *only* dequeue routed trace events, and these can be either analysed or relayed (*e.g.* the branching statement between lines 24 and 31 in lst. 3). By contrast, tracers in direct mode can relay events that have been routed their way, *but also* start routing trace events themselves when these are to be handled by other tracers.

Transitioning safely between tracing modes. A *router tracer* is one that currently receives events from a system process that is configured to be tracked by another tracer; the latter tracer must be in priority mode. In fig. 4b, T_P is the router tracer for T_Q, since Q (originally set to be traced by T_Q) shares T_P with process P once forked in fig. 4a, following assumption A_8. Similarly, T_P is also the router tracer for T_R. Our tracer choreographies observe the invariant that every tracer in priority mode has exactly one router tracer. Moreover, if any other tracer along the path between this tracer and the router tracer is also in priority mode, it must share the same router tracer.



(a) T_Q observes event `ext` before T_P routes `rcv`

(b) T_Q processes priority events routed by T_P first

Fig. 5. Trace event order preservation using priority (•) and direct (∘) tracer modes (analysers omitted)

```
def $\textsc{Loop}_\bullet(\sigma, p_{\mathrm{a}})$
  forever do
    # Trace event messages collected directly are
    # left in the queue to be handled in $\circ$ mode
    $r \leftarrow$ next rtd message from queue K
    $m \leftarrow r.\mathrm{emb}$
    if $m.\mathrm{type} = \mathrm{evt}$ then
      $\sigma \leftarrow \textsc{HandleEvent}_\bullet(\sigma, r, p_{\mathrm{a}})$
    else if $m.\mathrm{type} = \mathrm{dtc}$ then
      # dtc command routed back from ancestor
      $\sigma \leftarrow \textsc{HandleDtc}(\sigma, r, p_{\mathrm{a}})$
    end if
  end forever
end def
```

```
def $\textsc{HandleEvent}_\bullet(\sigma, r, p_{\mathrm{a}})$
  $e \leftarrow r.\mathrm{emb}$
  if $e.\mathrm{act} = \mathrm{frk}$ then
    $\sigma \leftarrow \textsc{HandleFork}_\bullet(\sigma, r, p_{\mathrm{a}})$
  else if $e.\mathrm{act} = \mathrm{ext}$ then
    $\sigma \leftarrow \textsc{HandleExit}_\bullet(\sigma, r, p_{\mathrm{a}})$
  else if $e.\mathrm{act} \in \{\mathrm{snd}, \mathrm{rcv}\}$ then
    $\textsc{HandleComm}_\bullet(\sigma, r, p_{\mathrm{a}})$
  end if
end def
```

```
def $\textsc{HandleFork}_\bullet(\sigma, r, p_{\mathrm{a}})$
  $e \leftarrow r.\mathrm{emb}$
  if $(p_{\mathrm{t}} \leftarrow \sigma.\Pi(e.\mathrm{src})) \neq \bot$ then
    $\textsc{Relay}(r, p_{\mathrm{t}})$
    $\sigma.\Pi \leftarrow \sigma.\Pi \cup \{\langle e.\mathrm{tgt}, p_{\mathrm{t}} \rangle\}$
  else
    Monitor $p_{\mathrm{a}}$ analyses event $e$
    $p'_{\mathrm{t}} \leftarrow r.\mathrm{rtr}$
    $\sigma \leftarrow \textsc{Instrument}_\bullet(\sigma, e, p'_{\mathrm{t}})$
  end if
  return $\sigma$
end def
```

```
def $\textsc{HandleExit}_\bullet(\sigma, r, p_{\mathrm{a}})$
  $e \leftarrow r.\mathrm{emb}$
  if $(p_{\mathrm{t}} \leftarrow \sigma.\Pi(e.\mathrm{src})) \neq \bot$ then
    $\textsc{Relay}(r, p_{\mathrm{t}})$
  else
    Monitor $p_{\mathrm{a}}$ analyses event $e$
    # Remove terminated process $e.\mathrm{src}$ from group
    $\sigma.\Gamma \leftarrow \sigma.\Gamma \setminus \{\langle e.\mathrm{src}, \bullet \rangle\}$
    $\textsc{TryGC}(\sigma, p_{\mathrm{a}})$
  end if
  return $\sigma$
end def
```

```
def $\textsc{HandleComm}_\bullet(\sigma, r, p_{\mathrm{a}})$
  $e \leftarrow r.\mathrm{emb}$
  if $(p_{\mathrm{t}} \leftarrow \sigma.\Pi(e.\mathrm{src})) \neq \bot$ then
    $\textsc{Relay}(r, p_{\mathrm{t}})$
  else
    Monitor $p_{\mathrm{a}}$ analyses event $e$
  end if
end def
```

```
Expect: $r.\mathrm{emb.iss} = \mathrm{self}() \vee \sigma.\Pi(r.\mathrm{emb.tgt}) \neq \bot$
def $\textsc{HandleDtc}(\sigma, r, p_{\mathrm{a}})$
  $c \leftarrow r.\mathrm{emb}$
  if $(p_{\mathrm{t}} \leftarrow \sigma.\Pi(c.\mathrm{tgt})) \neq \bot$ then
    $\textsc{Relay}(r, p_{\mathrm{t}})$
  else
    # Mark process $c.\mathrm{tgt}$ in group as detached
    $\sigma.\Gamma \leftarrow \sigma.\Gamma \setminus \{\langle c.\mathrm{tgt}, \bullet \rangle\}$
    $\sigma.\Gamma \leftarrow \sigma.\Gamma \cup \{\langle c.\mathrm{tgt}, \circ \rangle\}$
    $\gamma = \{\langle p_{\mathrm{s}}, d \rangle \mid \langle p_{\mathrm{s}}, d \rangle \in \sigma.\Gamma, d = \bullet\}$
    if $\gamma = \emptyset$ then
      $\textsc{Loop}_\circ(\sigma, p_{\mathrm{a}})$ # Switch tracer to $\circ$ mode
    end if
  end if
  return $\sigma$
end def
```

Lst. 3. Tracer loop that handles priority trace events and message relaying

A tracer in priority mode coordinates with its router tracer to determine whether all of the events for its tracked system processes have been routed. A tracer must effect this procedure for every process it currently tracks, recorded in Γ, before it can safely transition to direct mode, and start operating on the trace events it collects directly. The tracer issues a special detach command message, dtc, to notify the router tracer that it is now responsible for tracing a particular system process. Detach commands contain the PIDs of the issuer tracer in priority mode and system process in question, accessed via the fields iss and tgt respectively, described in tbl. 5 in app. A.

Fig. 5b shows tracer T_Q in priority mode (●) sending the command $\langle \mathrm{dtc}, \mathrm{q}_{\mathrm{t}}, \mathrm{q}_{\mathrm{s}} \rangle$ for Q, step ⑬, after it starts tracing this process directly, step ⑫. This transaction is implemented by Detach in lst. 6 of app. A, where $\langle \mathrm{dtc}, p'_{\mathrm{t}}, p_{\mathrm{s}} \rangle$ is sent to the router tracer p_{t} on line 10, after it invokes Preempt. In fig. 5b, dtc issued by T_Q follows `rcv` and `frk` in the message queue of tracer T_P. The router tracer processes its message queue sequentially in steps ⑩, ⑰, ⑲, ⑳ and ㉘. These trace events are forwarded to neighbouring tracers as necessary in steps ⑱ and ㉑ (see lines 3–5 in lst. 1). It also routes the dtc command back to the issuer tracer in step ㉙ where, once handled, marks the system process as *detached* from the router tracer. HandleDtc in lst. 3 effects this update on the routing map $\Gamma : \textsc{Pid}_\mathsf{s} \rightharpoonup \{\circ, \bullet\}$ of the issuer tracer on lines 58 and 59. Once all the processes in Γ become detached, the tracer transitions to direct mode by executing $\textsc{Loop}_\circ$; this check is performed on lines 60 and 61 in lst. 3. While in priority mode, T_Q handles the prioritised ($\bullet$) events forwarded by T_P in the *correct order* stipulated earlier in fig. 2b (steps ㉓ and ㉔). This is followed by handling the command dtc in step ㉚. The transition from priority to direct mode for T_Q in fig. 5b takes place in step ㉛. Finally, the trace event `ext` is handled in the correct order in step ㉜ (as opposed to step ⑱ in fig. 5a).

A detach command c originating at the router tracer may be relayed through multiple intermediate tracers until it reaches its destination. Every intermediate tracer purges the association $c.\mathrm{tgt} \mapsto p_\mathrm{t}$ from its routing map Π for some neighbouring tracer PID p_t. This functionality is provided by RelayDtc and RouteDtc in lst. 1: despite their similar logic, RouteDtc is used by the tracer to commence the routing of detach commands, whereas RelayDtc merely forwards commands to other tracers. While these steps are not shown in fig. 5b, we briefly remark that tracer T_P would remove from Π the mapping $\mathsf{q}_\mathsf{s} \mapsto \mathsf{q}_\mathsf{t}$, calling RouteDtc to start routing back the detach command $\langle \mathsf{dtc}, \mathsf{q}_\mathsf{t}, \mathsf{q}_\mathsf{s} \rangle$ it receives from T_Q. In due course, T_P also removes $\mathsf{r}_\mathsf{s} \mapsto \mathsf{q}_\mathsf{t}$ for process R once it handles $\langle \mathsf{dtc}, \mathsf{q}_\mathsf{t}, \mathsf{r}_\mathsf{s} \rangle$ sent by tracer T_R. When it receives the routed detach command $\langle \mathsf{rtd}, \mathsf{p}_\mathsf{t}, \langle \mathsf{dtc}, \mathsf{q}_\mathsf{t}, \mathsf{r}_\mathsf{s} \rangle \rangle$ from T_P, T_Q removes $\mathsf{r}_\mathsf{s} \mapsto \mathsf{r}_\mathsf{t}$ from Π and relays it, in turn, to tracer T_R using RelayDtc.

Selective instrumentation. In practice, one might want to have the flexibility to *group processes* under a single monitor to analyse them as one *component*. Our algorithm selectively instruments (new) tracers for particular system processes using the map, $\Phi : \textsc{Sig} \rightharpoonup \textsc{Mon}$: it maps the code signatures, g (of the system process forked), to the monitoring code, v (to be executed by the newly spawned monitor). Instrument in lst. 2 applies Φ to the code signature, where $e.\mathrm{sig} = g$, in the fork event e on lines 3 and 13. When $\Phi(g) = \bot$, instrumentation is not performed, and the tracer is automatically shared by the new process $e.\mathrm{tgt}$, according to assumptions A_7 and A_8.

Garbage collection. Our outline set-up can shrink in size by discarding tracers that are no longer needed. A tracer self-terminates after its routing map Π *and* tracked-processes map Γ become empty; this check is performed by TryGC in lst. 6 in app. A. The tracer purges process references from Γ when it handles exit trace events via $\textsc{HandleExit}_\circ$ and $\textsc{HandleExit}_\bullet$ (lsts. 1 and 3). Note that, even when $\Gamma = \emptyset$ and the tracer has no processes to analyse, it might still be required to route trace events to adjacent tracers, *i.e.,* $\Pi \neq \emptyset$. The garbage collection check is therefore performed each time mappings from Π *or* Γ are removed on lines 39, 54 and 72 in lst. 1, and line 41 in lst. 3. In fig. 5b, tracer T_Q would terminate sometime after handling the exit event `ext` for process Q in step ㉜, once the routed detach command $\langle \mathsf{rtd}, \mathsf{p}_\mathsf{t}, \langle \mathsf{dtc}, \mathsf{q}_\mathsf{t}, \mathsf{r}_\mathsf{s} \rangle \rangle$ it receives from T_P is relayed to tracer T_R.

4 EVALUATION

We give a comprehensive evaluation to assess quantitative aspects of inline and outline monitoring. Our evaluation takes the form of a case study that instantiates the monitoring problem from sec. 3 to a RV setting, where event streams are analysed to reach acceptance/rejection verdicts in connection to satisfactions/violations of correctness properties [Bartocci et al. 2018; Francalanza et al. 2017]. The set-up follows that of fig. 1, where the analysis components (used in fig. 1 and in inlined monitors) are synthesised from syntactic descriptions of the properties of interest. Our synthesis compiles properties down to *automata-based* monitors following [Aceto et al. 2019a]. We evaluate the different approaches in terms of runtime overheads and, by this, asses their viability. We follow an approach similar to [Bartocci et al. 2019], and consider the following overhead performance metrics: *(i)* mean scheduler utilisation, as a percentage of the total available capacity, *(ii)* mean memory consumption, measured in GB, *(iii)* mean round trip time (RTT), measured in milliseconds (ms), and, *(iv)* mean execution duration, measured in seconds (s). Our measurements are collected *globally* by sampling the runtime environment in which the SuS and monitoring system execute.

4.1 Scope

Our evaluation focusses on master-slave systems [Tarkoma 2010], where one central process, called the *master*, creates and farms out *tasks* to multiple *slave* processes. This class of decentralised systems satisfies assumptions A_1–A_5 from sec. 3, and is used pervasively in areas such as DNS, IoT and Big Data. On a local concurrency scale, master-slave systems underlie applications like thread pools and web servers.[1] We require an evaluation setup that targets the *instrumentation layer* of the various monitoring approaches. To achieve this, we need correctness properties that, while being parametric w.r.t. components, yield an analysis component that is uniform across the various approaches. We opted *not* to use global properties because, in a decentralised setting such as ours, individual monitors would need to cooperate in order to reach global verdicts. Decentralised inline and outline monitors interact in fundamentally different ways (*e.g.* inline monitors typically can query the internal data structures of the SuS whereas outline monitors would need to replicate this state externally) and this discrepancy introduces runtime biases that make the results hard collect and interpret. Instead, our evaluation exclusively employs *local* properties where the synthesised analysis components can reach verdicts without the need to interact with other analysing components [Attard and Francalanza 2017; Chen and Rosu 2009; Jin et al. 2012; Neykova and Yoshida 2017a; Reger et al. 2015]; this fixes the analysis overhead parameter to a uniform *constant* across all experiments. In fact, our synthesised analysis is identically pluggable in both the inline and outline monitoring algorithms. The properties we use for benchmarks translate to monitors that loop continually in order that these exert the *maximum level* of overhead possible.

4.2 Methodology

We use Erlang [Armstrong 2007] to implement our evaluation set-up and monitoring algorithms. Erlang adopts the actor model of computation [Agha et al. 1997], implementing them as *lightweight* processes. Actors interact via *asynchronous* messaging, changing their (local) internal state based on messages received. Every actor owns a message queue, called the *mailbox*, where messages can be taken out-of-order. Actors can also *fork* other actors to execute independently in their own process space. Every actor is identified via a PID that is assigned to it when forked. We use the term actor and process interchangeably in the rest of this section.

Implementation. The *inline monitoring* tool we developed for this study assumes access to the SuS source code. It instruments monitoring instructions into the target system via *code injection* by manipulating its parsed abstract syntax tree. The *modified* syntax tree is then compiled into an executable form, and the instrumented instructions perform the runtime analysis in a synchronous manner as the SuS executes. Our implementation of the *outline monitoring* algorithm in sec. 3 maps tracer processes to Erlang actors. Tracers collect the trace events by leveraging the *native* tracing infrastructure provided by the Erlang Virtual Machine (EVM). This infrastructure complies with assumptions A_6–A_8. EVM tracing directs trace event messages from system processes to tracer mailboxes acting as the tracer messages queues K of sec. 3.2. The maps Π, Φ and Γ are implemented using Erlang maps for efficient access. We implement the two trace analysis variants of fig. 1. For the arrangement in fig. 1a, the analysis is forked as a separate actor where tracers forward their event messages. Line 4 in lst. 6 of app. A indicates the point at which the actor tasked with the analysis is created whereas line 14 signals said actor to terminate when garbage collection takes place. The analysis is incorporated directly into tracers for the merged monitor case in fig. 1b.

---

[1] We could have employed a peer-to-peer set-up, but this complicates the evaluation considerably.

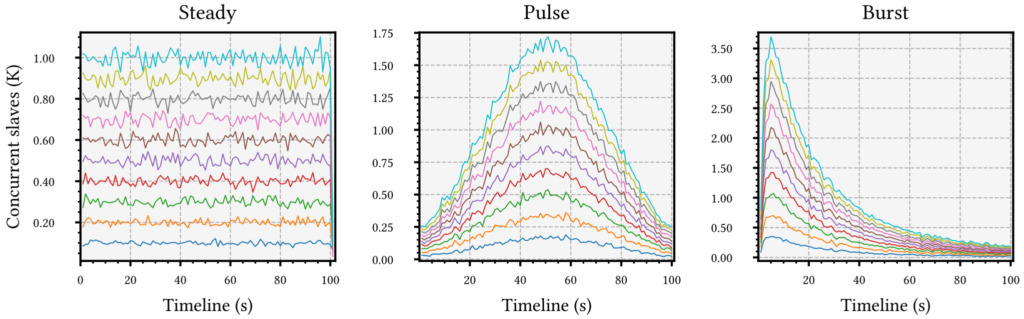


Fig. 6. Steady, Pulse and Burst load distributions with 100 k slaves for the duration of 100 s

The SuS. We opt for a custom-built evaluation platform that emulates models of master-slave systems. The decision *not* to go with off-the-shelf (*e.g.* web servers, thread pools, *etc.*) systems stems from three core drawbacks these have, namely: *(i)* they make it challenging to *precisely control* particular experiment parameters conducive to *repeatable results*, *(ii)* do not provide hooks that permit *accurate* measurement taking, *(iii)* often embody *highly-specific* use cases that make it difficult to generalise the findings obtained. Our evaluation platform is *parametrisable* to emulate different system models. The *tasks* farmed out by the master consist of *work requests* that a slave receives, processes and echoes back. A slave is set to terminate once all of its work requests have been handled and acknowledged by the master. The parameter w in our framework regulates the number of work requests that can be batched in one task; the *actual* amount of work requests *per slave* is drawn randomly from a normal distribution with mean $\mu = w$, and a standard deviation $\sigma = w \times 0.02$. This ensures a degree of variability in the amount of messages exchanged between the master and each slave. The speed with which the system reacts to work requests can be controlled via the parameters $\Pr(send)$ and $\Pr(recv)$. To distribute tasks uniformly amongst slaves, the master interleaves the sending and receiving of work requests: $\Pr(send)$ and $\Pr(recv)$ can bias this behaviour. $\Pr(send)$ determines the probability that a work request is assigned by the master to a slave. $\Pr(recv)$ controls the probability that a work request received by the master is handled and acknowledged. Load on the system is induced by the master when it creates slave processes; the total number of slaves that are created during one experiment is set using the parameter n.

Load models. Our system considers three load shapes (fig. 6) that establish how the creation of slaves based on the parameter n is distributed along the *load timeline* t. The load timeline is represented as a sequence of discrete *logical time units* that denote instants at which a new set of slaves is created by the master. *Steady* loads reproduce executions where a system operates under stable conditions. These are modelled on a homogeneous Poisson distribution with *rate* λ, specifying the mean number of slaves that are created at every time instant along the load timeline with duration $t = \lceil n/\lambda \rceil$. *Pulses* emulate scenarios where a system undergoes gradually-increasing load peaks. The pulse load shape is parametrised by t and the spread, s, that controls how slowly or sharply the system load increases as it approaches its peak halfway along t. Pulses follow a normal distribution with $\mu = t/2$ and $\sigma = s$. *Burst* loads capture scenarios where a system is stressed due to instant load spikes: these are based on a log-normal distribution with $\mu = \ln(m^2/\sqrt{p^2 + m^2})$, $\sigma = \sqrt{ln(1 + p^2/m^2)}$ where $m = t/2$ and p is the pinch controlling the intensity of the initial load burst.

Experiment set-up. To meet the objectives set out in this section, we conduct two case studies where the SuS is configured with n = 10k for *moderate* loads and n = 100k *high* loads. The number of work requests per task is set to $w = 100$. Pr(*send*)=Pr(*recv*)=0.9 fixes the probability of sending and acknowledging work requests: this emulates a system that reacts promptly to load, but at the same time, exhibits slight processing delays that arise in a master-slave architecture. Our chosen parameter values instantiate the SuS to model *realistic* web response time where the request intervals observed at the server follow a Poisson process [Ciemiewicz 2001; Kayser 2017; Liu et al. 2001]. Further detail regarding the validation of this model are given in app. B. For these experiments, the total loading time is set to $t = 100$s. We use the term *experiment* to denote a series of ten *benchmarks* where the SuS is configured with one particular monitoring set-up (*e.g.* with outline monitors). Load to the set-up is added *incrementally* at each benchmark until the maximum load is reached, *e.g.* for the case study with $n = 10$k slaves, we start with the first benchmark set to $n_1 = 1$k and progress to $n_{10} = 10$k in steps of 1 k. We repeated *ten* readings for each experiment, and aggregated the results by computing the weighted mean for the performance metrics mentioned above. Consult app. B for the full list of precautions. The experiments were conducted on an Intel Core i7 M620 64-bit machine with 8GB of memory, running Ubuntu 18.04 and Erlang/OTP 22.2.1.

4.3 Results and Discussion

Our results are reported in tbls. 1 and 2 and figs. 7–10, plotting each performance metrics (y-axis) against the slave processes (x-axis) for every monitoring mode; the *unmonitored* system is inserted as a baseline reference. Fitted data plots corresponding to figs. 7–10 are given in app. C.

Moderate loads. Our first batch of results considers loads that are slightly higher than those employed by the state-of-the-art to evaluate decentralised, concurrent and distributed runtime monitoring, *e.g.* [Attard and Francalanza 2017; Berkovich et al. 2015; Cassar and Francalanza 2016; Colombo and Falcone 2016; El-Hokayem and Falcone 2017; Francalanza and Seychell 2015; Mostafa and Bonakdarpour 2015; Neykova and Yoshida 2017a,b; Scheffel and Schmitz 2014]; works like [Chen and Rosu 2007, 2009; Reger et al. 2015] consider higher loads, but they evaluate sequential monitoring. Crucially, neither of the aforementioned studies employ different load shapes in their analysis: they either use loads modelled on a Poisson process, *i.e.,* Steady load, or do not specify the load types considered. The SuS set with $n = 10$k slaves and $w = 100$ work requests per slave generates $\approx n \times w \times$(work requests and responses)=2M messages exchanged between the master and slaves, producing 2M × (`snd` and `rcv` trace events) = 4M trace events. Tbl. 1 reports the percentage overhead at n=10k. It shows that inline and the two flavours of outline monitoring induce negligible execution slowdown for all three load shapes (*e.g.* 0.77 % maximum for DM under Burst load); the memory consumption overhead behaves similarly. At the Steady load illustrated in fig. 7, memory consumption and RTT (round trip) grow linearly in the number of slave processes. Under the Pulse and Burst loads in tbl. 1, inline monitoring induces negligible scheduler overhead. This is markedly higher for outline monitoring (DS and DM), mostly caused by the dynamic reconfiguration of the

	Steady			Pulse			Burst		
Filler	I	DS	DM	I	DS	DM	I	DS	DM
Scheduler	1.68	42.13	22.60	1.54	35.17	18.12	2.08	38.92	26.03
Memory	0.01	0.10	0.09	0.01	0.08	0.04	0.03	0.10	0.06
RTT	4.37	67.05	58.36	7.72	82.85	60.79	20.17	859.91	666.46
Execution	0.09	0.40	0.28	0.10	0.32	0.22	0.12	1.09	0.77

Inline (I), Decentralised outline separate (DS), Decentralised outline merged (DM)

Tbl. 1. Percentage runtime overhead taken at the maximum load $n = 10$k

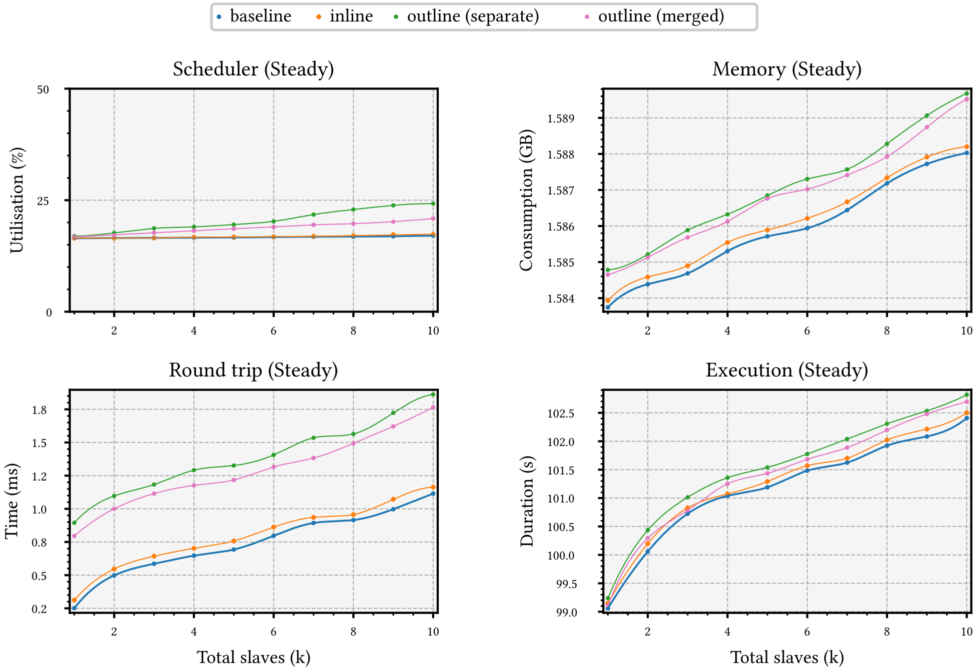


Fig. 7. Mean runtime overhead for monitoring the master and slave processes (10 k slaves)

monitoring choreography. Tbl. 1 also suggests that the RTT is very sensitive to the type of load applied, and it increases for the load shapes Steady, Pulse and Burst respectively. In fact, the latter load shape induces a sharp growth in the RTT for outline monitoring at around 9k ~ 10k slaves, as illustrated in fig. 8. This indicates that specific load shapes prompt very different behaviours from the monitors, and should be taken into account.

Despite the clear discrepancies (percentage-wise) in scheduler and RTT overheads between inline and outline monitoring in tbl. 1, these are *comparable* (value-wise) for the loads that are typically used in other bodies of work, as shown in figs. 7 and 8 (*e.g.* the worst discrepancy is 11ms for RTT under Burst). Merging the analysis with tracing as in fig. 1b yields improvements, but its effect is negligible. For certain performance metrics, our data plots do not allow us to confidently extrapolate our results. A case in point is the RTT Burst plot for outline monitoring, which raises the question of whether the trend remains consistent when the number of slaves exceeds 10k.

High loads. We increase the number of slaves to n=100k and keep w=100, to generate 20 M messages and 40 M trace events. Our aim is to assess how the monitored system performs *under stress*, and whether this reveals aspects that do not emerge at lower loads. Since these loads span a broader range, this also gives us a reasonable level of confidence when extrapolating our observations. Particular, we also include the measurements obtained for the SuS with a *centralised* monitoring set-up for this case study, to better isolate the effects of outline monitoring.

Tbl. 2 confirms that inline monitoring induces lower overheads. However, dissecting these results uncovers a few surprising aspects. For instance, the memory overhead between inline and outline monitoring with the separate analysis is 13.3 % under a Steady load at its highest point of 100 k slaves. This overhead is arguably tolerable for a number of applications. When merging outline tracing with its analysis as in fig. 1b, this discrepancy goes down to respectable 6.8 %. Centralised

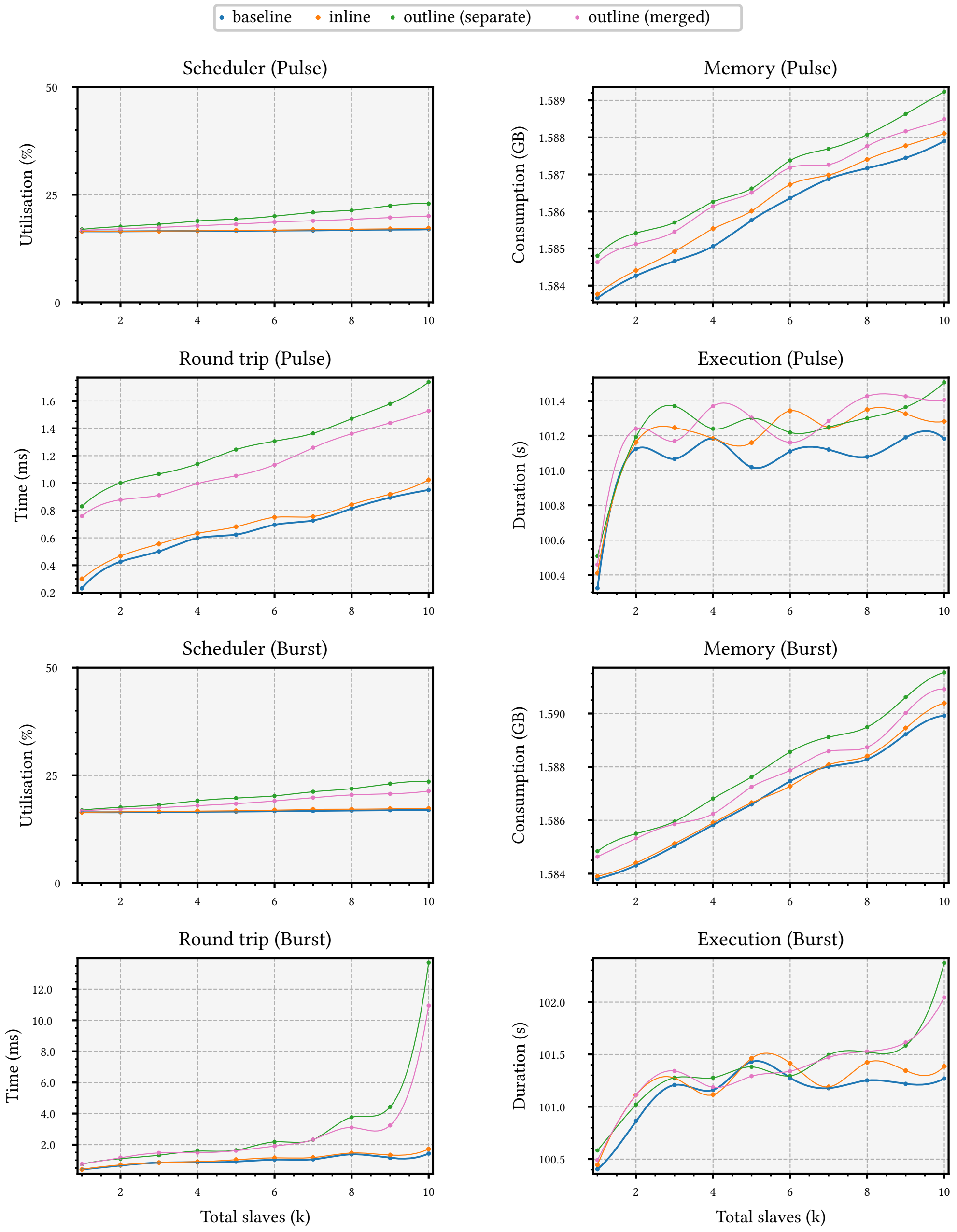


Fig. 8. Mean runtime overhead for monitoring the master and slave processes (10 k slaves, cont.)

outline monitoring further lower this difference to a negligible 0.6 %; this seems to debunk the general assumption that outline monitoring necessarily leads to infeasible overheads. The plots in fig. 9 also show that under Steady loads, the overhead for the memory, RTT and execution duration are comparable to inline monitoring up to the considerable load of around 40 k slaves (*i.e.,* 8 M

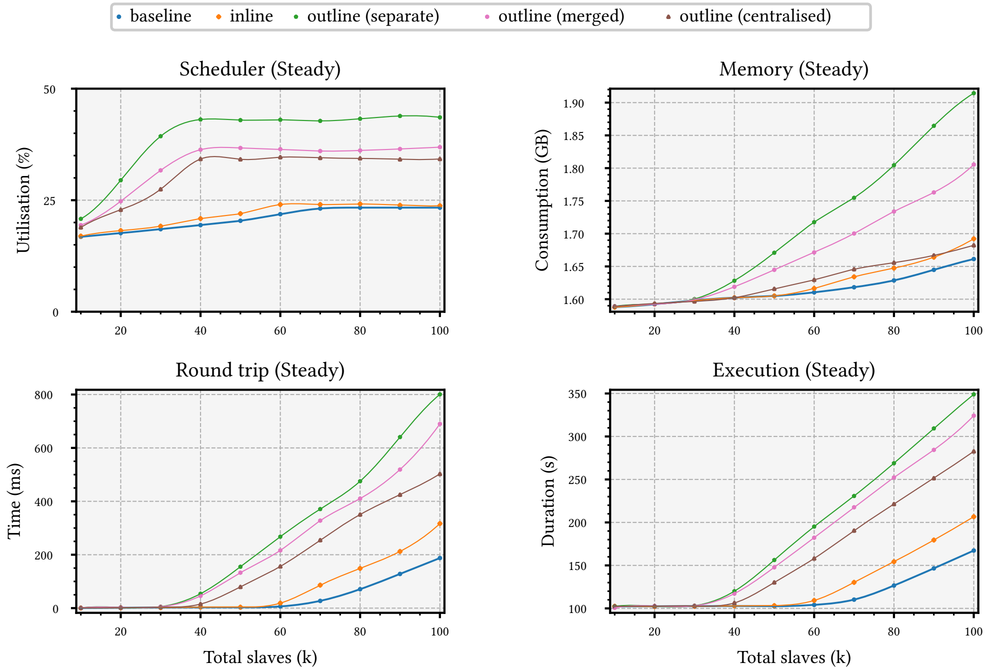


Fig. 9. Mean runtime overhead for monitoring the master and slave processes (100 k slaves)

messages and 16 M trace events). Tbl. 2 indicates that the RTT overhead for outline monitoring *decreases* for the load shapes Steady, Pulse and Burst respectively; this trend is also mirrored in the execution slowdown metric. These results contrast with the ones in tbl. 1, where the overhead for said metrics *gradually increases* under the same load shapes. This suggests that outline monitoring exhibits a degree of robustness at high numbers for loads like Pulse and Burst, whose shapes induce higher stress in the SuS in comparison to consistent loads. In these two instances, outline monitoring pays a price in terms of memory overhead (tbl. 1), although the *maximum* overhead reported in our results, 24.6 %, may be acceptable for many scenarios. Merging the analysis with tracing in outline monitoring *consistently* yields lower overheads when compared to the variant with separate analysis, irrespective of the load shape.

We draw attention to the charts in figs. 9 and 10, where the memory consumption plot for centralised outline monitoring crosses over that of inline monitoring. This behavior emerges

	Steady				Pulse				Burst			
	I	DS	DM	C	I	DS	DM	C	I	DS	DM	C
Scheduler	1.8	86.8	58.1	46.8	2.9	85.5	55.6	47.5	3.1	84.4	57.9	51.6
Memory	1.9	15.2	8.7	1.3	2.9	18.1	11.8	2.2	3.1	24.6	15.4	1.9
RTT	68.9	326.9	267.9	167.9	72.7	257.8	238.7	189.0	28.4	120.6	114.3	100.1
Execution	23.5	108.6	93.8	68.9	24.5	101.8	93.5	72.8	15.7	82.0	77.5	69.3

Inline (I), Decentralised outline separate (DS), Decentralised outline merged (DM), Centralised outline (C)

Tbl. 2. Percentage runtime overhead taken at the maximum load $n = 100\text{k}$

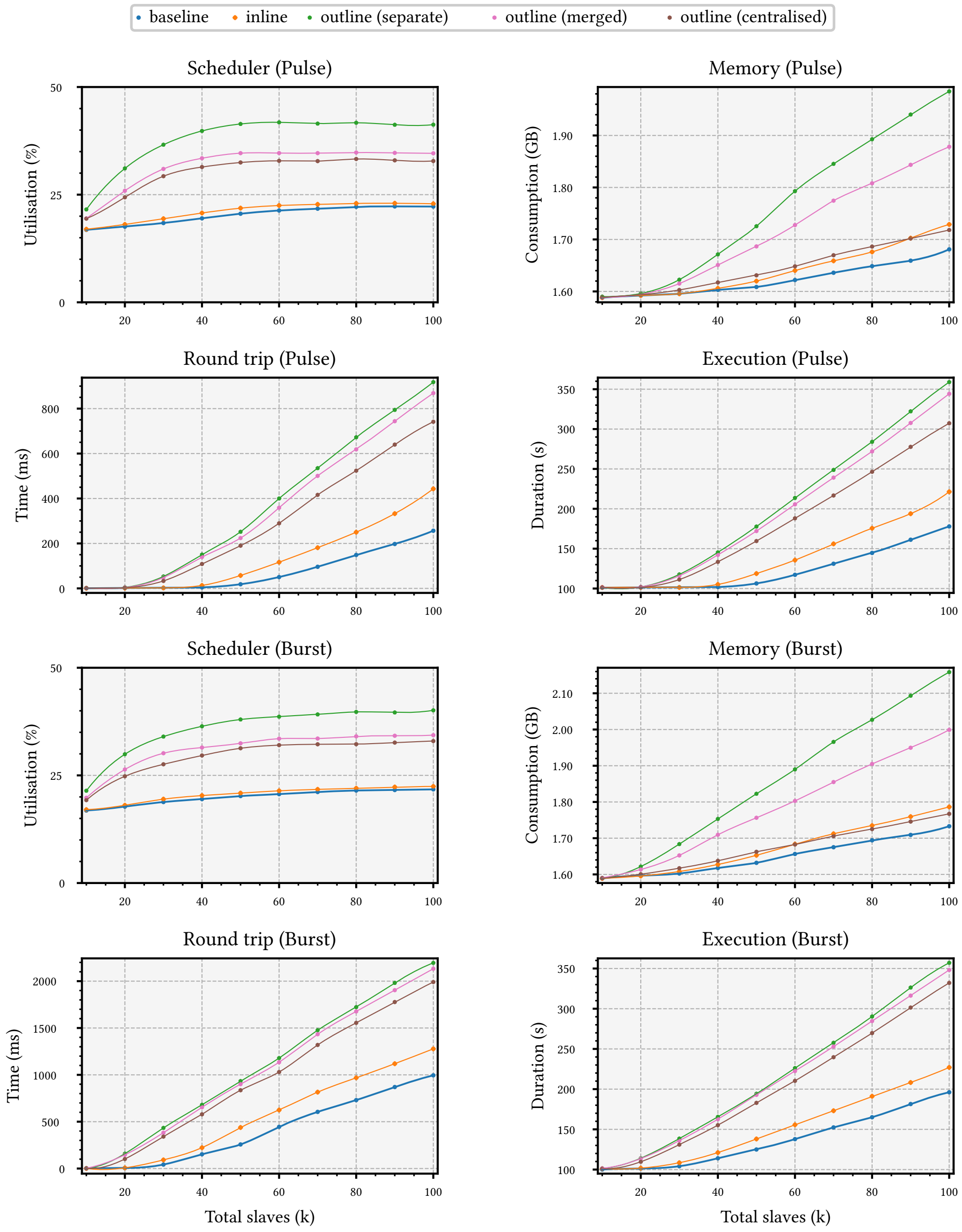


Fig. 10. Mean runtime overhead for monitoring the master and slave processes (100 k slaves, cont.)

because the former method consumes less memory than inline monitoring *on average*, but it then executes for a longer period of time. Figs. 9 and 10 illustrate the mean measurements obtained per experiment; a different depiction that shows the *total memory consumed* during the experiment can be found in app. C. From the figures reported in tbl. 2, one could even make a case that for

	Steady				Pulse				Burst			
	DS$_\text{m}$	DM$_\text{m}$	OS$_\text{s}$	DM$_\text{s}$	DS$_\text{m}$	DM$_\text{m}$	DS$_\text{s}$	DM$_\text{s}$	DS$_\text{m}$	DM$_\text{m}$	DS$_\text{s}$	DM$_\text{s}$
Scheduler	20.4	10.1	0.001	0.001	21.4	14.4	0.001	0.000	23.2	15.4	0.001	0.000
Memory	1.4	0.7	0.000	0.000	2.0	1.5	0.000	0.000	1.4	0.9	0.000	0.000
RTT	194.6	134.4	0.001	0.001	200.5	185.4	0.001	0.001	87.2	72.7	0.000	0.000
Execution	74.9	61.1	0.000	0.000	79.2	73.7	0.000	0.000	60.7	50.3	0.000	0.000

Decentralised outline separate on master (DS$_\text{m}$), Decentralised outline merged on master (DM$_\text{m}$)
Decentralised outline separate per slave (DS$_\text{s}$), Decentralised outline merged per slave (DM$_\text{s}$)

Tbl. 3. Percentage amortised runtime overhead on each slave taken at the maximum load $n = 100$k

settings where memory is limited but execution time is not, *outlined* centralised monitoring is more appropriate than *inlined* decentralised monitoring.

The memory consumption, RTT, and execution duration plots in figs. 9 and 10 exhibit a *linear growth* beyond specific x-axis thresholds. This contrasts with the plots in figs. 7 and 8 for $n = 10$k, where different trends may be observed: the execution duration plot under a Steady load shape grows (negative) quadratically in fig. 7, but follows a cubic trend in the case of Pulse and Burst loads in fig. 8; a similar effect is obtained in the RTT for Burst (consult the fitted data plots in app. C). These differences in runtime behaviour underscore the value of performing tests using reasonably high loads, as this increases the chances of *observing likely trends*. For instance, the empirical evidence obtained for moderate loads could mislead one to assert that outline monitoring scales very poorly in the case of RTT under moderately-sized Burst loads.

Estimated overhead on slaves. Our results thus far present an *overall view* of the overhead induced by runtime monitoring. In certain cases however, this measure is too coarse since we would be interested more in quantifying the overhead incurred at *each slave*: this bears particular relevance to distributed setting where the processing capability of the system is spread over heterogenous machines (*e.g.* in an IoT set-up deployed on edge nodes [Shi et al. 2016] with limited computing power, understanding slave overheads is essential). Our experiment set-up does not allow us to directly measure the overhead at each slave since our measurements are collected globally. Instead, Tbl. 3 shows the percentage overhead for decentralised outline monitoring induced on the master process *only*, together with the estimated *overhead apportioned* over each slave. The overhead incurred by the SuS when monitoring the master process, DS$_\text{m}$ and DM$_\text{m}$ in tbl. 3, is obtained by setting up the experiment with $n = 100$k and $w = 100$ as before. We can then approximate the *combined overhead* induced by the slaves by subtracting DS$_\text{m}$ and DM$_\text{m}$ from the *total overhead* obtained when the master and slaves are monitored together (DS and DM in tbl. 2). An *apportioned* overhead per slave can therefore be obtained by dividing this combined overhead by the number of slaves, *i.e.*, 100 k, to give DS$_\text{s}$ and DM$_\text{s}$ in tbl. 3. Figs. 18 and 19 in app. C show the gap in overhead between the SuS fixed with one monitor on the master process and to the fully-monitored system. The estimated figures in tbl. 3 clearly indicate that the two flavours of decentralised runtime monitoring from fig. 1 induce nominal overhead per slave for the load shapes we consider.

5 CONCLUSION

We provide a detailed study of asynchronous outline monitoring, an alternative to inline monitoring that is often discarded due to its high overheads. Our study makes a case that there are instances where outlining is the only available solution for analysing a system at runtime and that the overheads are tolerable in certain scenarios. To the best of our knowledge, the algorithm presented in sec. 3 differs from the state-of-the art in three fundamental ways: *(i)* it asynchronously gathers events from the SuS, *(ii)* effects the analysis using outline monitors, and, *(iii)* dynamically scales the runtime set-up as the SuS grows and shrinks. Our experiments in sec. 4 give scenarios that indicate the uses-cases where outline monitoring is best applied. They establish a pessimistic point of departure for outline monitoring: *(i)* RV monitoring typically does not exclude any events but less stringent analyses can resort to sampling to lower overheads [Sigelman et al. 2010]; *(ii)* RV analysis was carried out on every slave until termination, but RV verdicts may be reached early in the execution; We anticipate that less demanding settings such as general APM tools will lower further the overhead discrepancies reported.

5.1 Related Work

Decentralised monitoring in RV. Following standard texts on distributed computing [Buyya et al. 2011; Coulouris et al. 2005; Tarkoma 2010]. distributed systems are *necessarily* concurrent, but can be either centralised or decentralised, although the opposite *does not* always hold. Specifically, assumption A_1 in sec. 3 renders a system concurrent whereas lifting assumptions A_3 and A_4 changes this to distributed. Systems that rely on a global clock are neither concurrent nor distributed. For example, works such as [Colombo and Falcone 2016], while decentralised, do not qualify as distributed solutions; system components operate in synchronous rounds whereby a *unique* global trace can be reconstructed by combining the different traces collected at each component. Their approach does not address challenges such as message reordering (assumption A_3 in sec. 3).

Code injection is used in a number of tools targeting concurrent and distributed component systems. For example, [Sen et al. 2006] study decentralised monitors that are attached to different threads to collect and process trace events locally. In an earlier work [Sen et al. 2004], this investigation is conducted in a distributed setting using decentralised monitors that are weaved into components of the SuS. The authors focus on the efficiency of monitor communication but do not study nor quantify the overhead induced by runtime monitoring. Minimising overhead is also the focus of [Mostafa and Bonakdarpour 2015]. In this setting, the SuS consists of distributed asynchronous processes that communicate together via message-passing primitives over reliable channels. Similar to ours, their monitoring algorithm does not rely on a global notion of timing, and does not assume failing system components. The work by [Basin et al. 2015] is one of the few that considers distributed system monitoring where components and network links may fail. Despite the absence of a global clock, their monitoring algorithm is based on the timed asynchronous model for distributed systems [Cristian and Fetzer 1999] that assumes *highly-synchronised* physical clocks across nodes. In a different manner, [Bonakdarpour et al. 2016; Fraigniaud et al. 2014] address the problem of when the monitors themselves crash. Failure is an aspect that we do not presently address (see assumptions A_4 and A_5). The tools in [Basin et al. 2015; Cassar and Francalanza 2016; Jin et al. 2012] weave special instructions to enable the system to externalise its monitors, similar to fig. 1a. Crucially, inlining spares their algorithms from having to deal with reordered trace events.

Tools such as [Attard and Francalanza 2017; Neykova and Yoshida 2017a] target Erlang. In [Neykova and Yoshida 2017a], the authors propose a method that statically analyses the program communication flow, specified in terms of a multiparty protocol. Monitors attached to system processes check that the messages received coincide with the projected type, and in the case of failure, the associated processes are restarted. The authors show that their recovery algorithm induces less communication overhead, and improves upon the static process structure recovery mechanisms offered by the Erlang/OTP platform. Similarly, [Attard and Francalanza 2017] focus on decentralised outline monitoring in a concurrent setting. By contrast to [Neykova and Yoshida 2017a], they leverage the native tracing infrastructure offered by the EVM.

We remark that the works above all rely on bespoke evaluation platforms, making it hard to reproduce or directly compare their empirical results to ours. They either use loads modelled on a Poisson process, *i.e.,* Steady load, or fail altogether to specify the load types considered. Our empirical study has shown that different load shapes are indeed relevant.

Evaluation and benchmarking tools. Savina [Imam and Sarkar 2014] addresses the lack of a common benchmarking tool for actor-based systems. In the spirit of DaCapo [Blackburn et al. 2006], it provides a suite of diverse benchmarks that represent compute (as opposed to IO) intensive applications. These range from micro-benchmarks to classical concurrency problems and parallelism benchmarks. Similar our evaluation framework, Savina includes instantiations of the master-slave set-up (Trapezoidal Approximation and Precise Pi Computation), that are configurable with the number of slaves. However, our implementation accommodates more parameters: *(i)* the number of work requests per slave, *(ii)* the probability of allocating and acknowledging work requests, and, *(iii)* the type of load shape. In addition, we also support dynamic set-ups, as opposed to the static ones currently included in this suite. Savina measures the mean execution duration; we also collect the scheduler utilisation, memory consumption and RTT between the master and slaves. The requirement for a dynamic set-up, together with the performance metrics outlined in sec. 4 made Savina unsuitable to our use-case. Presently, Savina targets JVM actor-based languages. Like our implementation, Savina does not yet include benchmarks based on the peer-to-peer architecture.

Themis [El-Hokayem and Falcone 2017] is a tool that aims to facilitate the design and analysis of decentralised monitoring algorithms. It supports static set-ups where the number of system components and corresponding monitors is known and remains fixed at runtime. Unlike Savina or our benchmarking tool, Themis processes only pre-recorded tracers supplied via text files, making it incompatible with online monitoring. In [El-Hokayem and Falcone 2017], the authors claim that these trace files may be obtained from instrumented programs. Monitor and monitor communication in Themis is simulated via method calls that deposit messages inside blocking queues linked to each monitor. Like Savina, Themis is developed for Java applications.

The Behaviour, Interaction, Priority (BIP) framework models heterogenous real-time component systems. In BIP [Basu et al. 2006; El-Hokayem et al. 2018], the interaction between components is specified using syntactic descriptions that are parsed by a Java front-end and translated to C++ code. The automata-based operational model of BIP is implemented into their back-end platform that executes the generated code. BIP supports synchronous and asynchronous components that may run on the same or separate threads. While the back-end implementation relies on POSIX threads [Butenhof 1997] for easy integration with C++, this also limits the scalability of BIP when many asynchronous components are used, since each pthread takes kernel resources from the system. By contrast, the green processes used by Erlang allows our evaluation platform to scale considerably while incurring manageable overhead. Erlang process scheduling is performed by the EVM, making these much more lightweight when compared to pthreads. BIP is principally built as a flexible modelling tool that is inapplicable to our benchmarking requirements from sec. 4.

Kollaps [Gouveia et al. 2020] emulates distributed network conditions from an application-level perspective that considers the observable end-to-end properties such as latency or packet loss. The tool simplifies the network view by abstracting over the state of physical network appliances that sit in between nodes of the distributed application. Kollaps is fully-decentralised and agnostic of the application language and transport protocol. The authors show that Kollaps can closely model realistic network conditions. We plan to integrate Kollaps into our evaluation framework when extending it to account for further experiment variables such as packet loss and node failure (assumptions A_4 and A_5).

REFERENCES

Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. 2019a. Adventures in monitorability: from branching to linear time and back again. *Proc. ACM Program. Lang.* 3, POPL (2019), 52:1–52:29. https://doi.org/10.1145/3290365

Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. 2019b. An Operational Guide to Monitorability. In *SEFM (LNCS)*, Vol. 11724. Springer, 433–453. https://doi.org/10.1007/978-3-030-30446-1_23

Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. 1997. A Foundation for Actor Computation. *JFP* 7, 1 (1997), 1–72.

Nadia Alshahwan, Andrea Ciancone, Mark Harman, Yue Jia, Ke Mao, Alexandru Marginean, Alexander Mols, Hila Peleg, Federica Sarro, and Ilya Zorin. 2019. Some Challenges for Software Testing Research (Invited Talk Paper). In *ISSTA*. ACM, 1–3.

Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. 2016. Automating Failure Testing Research at Internet Scale. In *SoCC*. ACM, 17–28.

Joe Armstrong. 2007. *Programming Erlang: Software for a Concurrent World* (first ed.). Pragmatic Bookshelf.

Vinay Arora, Rajesh Kumar Bhatia, and Maninder Singh. 2016. A systematic review of approaches for testing concurrent programs. *CCPE* 28, 5 (2016), 1572–1611.

Duncan Paul Attard and Adrian Francalanza. 2017. Trace Partitioning and Local Monitoring for Asynchronous Components. In *SEFM (LNCS)*, Vol. 10469. Springer, 219–235.

Fred Baker. 1995. Requirements for IPv4 Routers. https://www.ietf.org/rfc/rfc1812.txt

Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. 2019. First international Competition on Runtime Verification: rules, benchmarks, tools, and final results of CRV 2014. *STTT* 21, 1 (2019), 31–70. https://doi.org/10.1007/s10009-017-0454-5

Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. In *Lectures on Runtime Verification*. LNCS, Vol. 10457. Springer, 1–33.

David A. Basin, Felix Klaedtke, and Eugen Zalinescu. 2015. Failure-Aware Runtime Verification of Distributed Systems. In *FSTTCS (LIPIcs)*, Vol. 45. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 590–603.

Ananda Basu, Marius Bozga, and Joseph Sifakis. 2006. Modeling Heterogeneous Real-time Components in BIP. In *SEFM*. IEEE Computer Society, 3–12.

Shay Berkovich, Borzoo Bonakdarpour, and Sebastian Fischmeister. 2015. Runtime Verification with Minimal Intrusion through Parallelism. *FMSD* 46, 3 (2015), 317–348.

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*. ACM, 169–190.

Borzoo Bonakdarpour, Pierre Fraigniaud, Sergio Rajsbaum, David A. Rosenblueth, and Corentin Travers. 2016. Decentralized Asynchronous Crash-Resilient Runtime Verification. In *CONCUR (LIPIcs)*, Vol. 59. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 16:1–16:15.

David R. Butenhof. 1997. *Programming with POSIX threads* (first ed.). Addison-Wesley.

Rajkumar Buyya, James Broberg, and Andrzej Goscinski. 2011. *Cloud Computing: Principles and Paradigms (Wiley Series on Parallel and Distributed Computing)*. Wiley.

Bryan Cantrill. 2006. Hidden in Plain Sight. *ACM Queue* 4, 1 (2006), 26–36.

Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. 2004. Dynamic Instrumentation of Production Systems. In *USENIX Annual Technical Conference, General Track*. USENIX, 15–28.

João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz. 2017. Chapter 5 - Source code transformations and optimizations. In *Embedded Computing for High Performance*, João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz (Eds.). Morgan Kaufmann, Boston, 137 – 183. https://doi.org/10.1016/B978-0-12-804189-5.00005-3

João M. P. Cardoso, José Gabriel F. Coutinho, Tiago Carvalho, Pedro C. Diniz, Zlatko Petrov, Wayne Luk, and Fernando M. Gonçalves. 2016. Performance-driven instrumentation and mapping strategies using the LARA aspect-oriented programming approach. *SPE* 46, 2 (2016), 251–287.

Ian Cassar and Adrian Francalanza. 2016. On Implementing a Monitor-Oriented Programming Framework for Actor Systems. In *IFM (LNCS)*, Vol. 9681. Springer, 176–192.

Francesco Cesarini and Simon Thompson. 2009. *Erlang Programming: A Concurrent Approach to Software Development* (first ed.). O'Reilly Media.

Feng Chen and Grigore Rosu. 2007. MOP: An Efficient and Generic Runtime Verification Framework. In *OOPSLA*. ACM, 569–588.

Feng Chen and Grigore Rosu. 2009. Parametric Trace Slicing and Monitoring. In *TACAS (LNCS)*, Vol. 5505. Springer, 246–261.

Kung Chen and Ju-Bing Chen. 2006. On Instrumenting Obfuscated Java Bytecode with Aspects. In *SESSICSE*. ACM, 19–26.

David M. Ciemiewicz. 2001. What Do You Mean? - Revisiting Statistics for Web Response Time Measurements. In *CMG*. Computer Measurement Group, 385–396.

Christian Colombo and Yliès Falcone. 2016. Organising LTL Monitors over Distributed Systems with a Global Clock. *FMSD* 49, 1-2 (2016), 109–158.

Christian Colombo, Gordon J. Pace, and Gerardo Schneider. 2009. LARVA—Safer Monitoring of Real-Time Java Programs (Tool Paper). In *SEFM*. IEEE Computer Society, 33–37.

George Coulouris, Jean Dollimore, and Tim Kindberg. 2005. *Distributed Systems: Concepts and Design* (fourth ed.). Addison Wesley.

Flaviu Cristian and Christof Fetzer. 1999. The Timed Asynchronous Distributed System Model. *IEEE Trans. Parallel Distrib. Syst.* 10, 6 (1999), 642–657.

Mathieu Desnoyers and Michel R. Dagenais. 2006. *The LTTng tracer : A low impact performance and behavior monitor for GNU / Linux.* Technical Report.

Antoine El-Hokayem and Yliès Falcone. 2017. Monitoring Decentralized Specifications. In *ISSTA*. ACM, 125–135.

Antoine El-Hokayem and Yliès Falcone. 2017. THEMIS: A Tool for Decentralized Monitoring Algorithms. In *ISSTA*. ACM, 372–375.

Antoine El-Hokayem and Yliès Falcone. 2020. On the Monitoring of Decentralized Specifications: Semantics, Properties, Analysis, and Simulation. *ACM Trans. Softw. Eng. Methodol.* 29, 1, Article 1 (Jan. 2020), 57 pages. https://doi.org/10.1145/3355181

Antoine El-Hokayem, Yliès Falcone, and Mohamad Jaber. 2018. Modularizing behavioral and architectural crosscutting concerns in formal component-based systems - Application to the Behavior Interaction Priority framework. *JLAMP* 99 (2018), 143–177. https://doi.org/10.1016/j.jlamp.2018.05.005

Úlfar Erlingsson. 2004. *The Inlined Reference Monitor Approach to Security Policy Enforcement.* Ph.D. Dissertation. Cornell University.

Úlfar Erlingsson and Fred B. Schneider. 1999. SASI Enforcement of Security Policies: A Retrospective. In *NSPW*. ACM, 87–95.

Pierre Fraigniaud, Sergio Rajsbaum, and Corentin Travers. 2014. On the Number of Opinions Needed for Fault-Tolerant Run-Time Monitoring in Distributed Systems. In *RV (Lecture Notes in Computer Science)*, Vol. 8734. Springer, 92–107.

Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. 2017. A Foundation for Runtime Monitoring. In *Runtime Verification (RV) (LNCS)*, Vol. 10548. Springer, 8–29. https://doi.org/10.1007/978-3-319-67531-2_2

Adrian Francalanza, Jorge A. Pérez, and César Sánchez. 2018. Runtime Verification for Decentralised and Distributed Systems. In *Lectures on Runtime Verification.* LNCS, Vol. 10457. Springer, 176–210.

Adrian Francalanza and Aldrin Seychell. 2015. Synthesising Correct Concurrent Runtime Monitors. *FMSD* 46, 3 (2015), 226–261.

Vijay K. Garg. 2015. *Elements of Distributed Computing* (first ed.). Wiley India.

Paulo Gouveia, João Neves, Carlos Segarra, Luca Liechti, Shady Issa, Valerio Schiavoni, and Miguel Matos. 2020. Kollaps: Decentralized and Dynamic Topology Emulation. In *EuroSys*. ACM, 23:1–23:16.

Duncan A. Grove and Paul D. Coddington. 2005. Analytical Models of Probability Distributions for MPI Point-to-Point Communication Times on Distributed Memory Parallel Computers. In *ICA3PP (LNCS)*, Vol. 3719. Springer, 406–415.

Christoph Heger, André van Hoorn, Mario Mann, and Dusan Okanovic. 2017. Application Performance Management: State of the Art and Challenges for the Future. In *ICPE*. ACM, 429–432.

Gregor Hohpe and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* (first ed.). Addison-Wesley Professional.

Shams Mahmood Imam and Vivek Sarkar. 2014. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *AGERE!@SPLASH*. ACM, 67–80.

Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov. 2018. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* 35, 3 (2018), 24–35.

Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Rosu. 2012. JavaMOP: Efficient Parametric Runtime Monitoring Framework. In *ICSE*. IEEE Computer Society, 1427–1430.

Nicolai M. Josuttis. 2007. *SOA in Practice: The Art of Distributed System Design: Theory in Practice* (first ed.). O'Reilly Media.

Bill Kayser. 2017. What Is the Expected Distribution of Website Response Times? https://blog.newrelic.com/engineering/expected-distributions-website-response-times

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-Oriented Programming. In *ECOOP (LNCS)*, Vol. 1241. Springer, 220–242.

Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. 2001. Java-MaC: a Run-time Assurance Tool for Java Programs. *Electr. Notes Theor. Comput. Sci.* 55, 2 (2001), 218–235.

Jay Ligatti, Lujo Bauer, and David Walker. 2005. Edit automata: enforcement mechanisms for run-time security policies. *International Journal of Information Security (IJIS)* 4, 1-2 (2005), 2–16. https://doi.org/10.1007/s10207-004-0046-8

Zhen Liu, Nicolas Niclausse, and César Jalpa-Villanueva. 2001. Traffic Model and Performance Evaluation of Web Servers. *Perform. Evaluation* 46, 2-3 (2001), 77–100.

Robert Martin. 2013. *Agile Software Development, Principles, Patterns, and Practices* (first ed.). Pearson.

Menna Mostafa and Borzoo Bonakdarpour. 2015. Decentralized Runtime Verification of LTL Specifications in Distributed Systems. In *IPDPS*. IEEE Computer Society, 494–503.

Rumyana Neykova and Nobuko Yoshida. 2017a. Let It Recover: Multiparty Protocol-Induced Recovery. In *CC*. ACM, 98–108.

Rumyana Neykova and Nobuko Yoshida. 2017b. Multiparty Session Actors. *Logical Methods in Computer Science* 13, 1 (2017).

Giles Reger, Helena Cuenca Cruz, and David E. Rydeheard. 2015. MarQ: Monitoring at Runtime with QEA. In *TACAS (LNCS)*, Vol. 9035. Springer, 596–610.

Richard J. Rossi. 2018. *Mathematical Statistics: An Introduction to Likelihood Based Inference.* Wiley.

César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, Ezio Bartocci, Domenico Bianculli, Christian Colombo, Yliès Falcone, Adrian Francalanza, Srdan Krstic, João M. Lourenço, Dejan Nickovic, Gordon J. Pace, José Rufino, Julien Signoles, Dmitriy Traytel, and Alexander Weiss. 2019. A survey of challenges for runtime verification from advanced application domains (beyond software). *FMSD* 54, 3 (2019), 279–335.

Torben Scheffel and Malte Schmitz. 2014. Three-Valued Asynchronous Distributed Runtime Verification. In *MEMOCODE*. IEEE, 52–61.

Fred B. Schneider. 2000. Enforceable Security Policies. *Transactions on Information and System Security (TISSEC)* 3, 1 (2000), 30–50. https://doi.org/10.1145/353323.353382

Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. 2004. Efficient Decentralized Monitoring of Safety in Distributed Systems. In *ICSE*. IEEE Computer Society, 418–427.

Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. 2006. Decentralized Runtime Analysis of Multithreaded Applications. In *IPDPS*. IEEE.

Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646.

Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP (LNCS)*, Vol. 4609. Springer, 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf

Connie U. Smith and Lloyd G. Williams. 2001. Software Performance AntiPatterns; Common Performance Problems and their Solutions. In *CMG§*. Computer Measurement Group, 797–806.

Connie U. Smith and Lloyd G. Williams. 2002. New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot. In *CMG*. Computer Measurement Group, 667–674.

Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual typing for first-class classes. In *OOPSLA*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 793–810. https://doi.org/10.1145/2384616.2384674

Sasu Tarkoma. 2010. *Overlay Networks: Toward Information Networking* (first ed.). Auerbach Publications.

Event	Action (e.act)	Field names	Description
fork	`frk`	src	PID p_s of the (parent) process invoking fork(g)
		tgt	PID p_s of the forked (child) process
		sig	Code signature g run by the forked process
exit	`ext`	src	PID p_s of the terminated process
send	`snd`	src	PID p_s of the sender process
		tgt	PID p_s of the recipient process
receive	`rcv`	src	PID p_s of the recipient process

Tbl. 4. Trace event messages data field names

A DECENTRALISED OUTLINE MONITORING ALGORITHM AUXILLARY CODE

Field access notation for tracer messages. Just as the message qualifier is accessible through the field name m.type, so are the data elements of the respective trace event types `frk`, `ext`, `snd`, and `rcv`. These are catalogued in tbl. 4.

Trace event acquisition. The tracing mechanism is defined by the operations Trace, Clear and Preempt listed in lst. 4. Trace enables a tracer p_t to register its interest in being notified about trace events of a system process p_s; this operation can be undone using Clear. Clear blocks the caller, and returns only once all the trace event messages for p_s that are in the process of being delivered are deposited into the message queue of p_t. Preempt combines Clear and Trace, enabling a tracer p_t to take over the tracing of process p_s from another tracer p'_t. The preemption instructions on lines 19–20 are ideally executed *atomically* to prevent potential trace event loss that could occur when switching tracers. This guarantee however, depends exclusively on the implementation of the underlying tracing mechanism. We recall that, following assumption A_8, tracing is *inherited* by every child process that a traced system process forks; Clear or Preempt can then be used to alter this arrangement.

Trace routing and relaying. Our algorithm performs routing using two operations, Route and Relay in lst. 5. Route creates a *new* message, r, with type rtd, that embeds trace events or dtc commands

```
 1 def TRACE($p_s$, $p_t$)
 2   if $p_s$ is not traced then
 3     Set the tracer for $p_s$ to $p_t$; $p_t$ will trace new
         descendant processes $p_{s_1}, p_{s_2}, \ldots$ forked
         by $p_s$ automatically (assumption $A_8$)
 4     while $p_s$'s tracer is set do
 5       $s \leftarrow$ read next event for $p_s$ from
           trace event source
 6       $e \leftarrow$ encode $s$ as a message
 7       $p_t\,!\,e$
 8     end while
 9   end if
10 end def

11 def CLEAR($p_s$, $p_t$)
12   if $p_s$ is traced then
13     Clear the tracer $p_t$ for $p_s$; $p_t$ still traces
         the descendant processes $p_{s_1}, p_{s_2}, \ldots$ of $p_s$
14     Block until the trace events for $p_s$ that are in
         transit are delivered to $p_t$
15   end if
16 end def

Expect: $p_s$'s tracer is set
17 def PREEMPT($p_s$, $p_t$)
18   $p'_t \leftarrow p_s$'s tracer
19   CLEAR($p_s$, $p'_t$)
20   TRACE($p_s$, $p_t$)
21 end def
```

Lst. 4. Trace event acquisition, clear, and preemption operations offered by the tracing mechanism

```
Expect: $m.\text{type} = \text{evt} \lor m.\text{type} = \text{dtc}$
def Route($m, p_t$)
  $p_t\ !\ \langle \text{rtd}, \text{self}(), m \rangle$
end def

Expect: $m.\text{type} = \text{rtd}$
def Relay($m, p_t$)
  $p_t\ !\ m$
end def
```

Lst. 5. Message routing and relaying operations

```
def Tracer($\sigma, v, p_s, p_t$)
  # Executable monitor map $\sigma.\Phi$ is copied to the
  # new (child) tracer state $\sigma'$; component group $\Gamma$
  # is initialised with the process being traced, $p_s$
  $\sigma' \leftarrow \langle \Pi \leftarrow \emptyset, \sigma.\Phi, \Gamma \leftarrow \{\langle p_s, \bullet \rangle\} \rangle$
  Detach($p_s, p_t$)
  $p_a \leftarrow \text{fork}(v)$ executable monitor
  # New tracer is started in $\bullet$ mode to process
  # routed events before locally-traced ones
  Loop$_\bullet$($\sigma', p_a$)
end def

def Detach($p_s, p_t$)
  $p'_t \leftarrow \text{self}()$
  Preempt($p_s, p'_t$)
  $p_t\ !\ \langle \text{dtc}, p'_t, p_s \rangle$
end def

def TryGC($\sigma, p_a$)
  if $\sigma.\Gamma = \emptyset \land \sigma.\Pi = \emptyset$ then
    Signal monitor $p_a$ to terminate
    Terminate tracer
  end if
end def
```

Lst. 6. Operations used by the (○) and priority (●) tracer loops

that need to be routed. The PID of the tracer process invoking Route is included into a routed message: we refer to this process as the *router tracer* that it is responsible for injecting the message into the tracer choreography. This PID is retrievable using the field m.rtr (see tbl. 5), and enables other tracers to identify the tracer that initiated the message dispatch. Routed messages can *only* be handled by tracers or forwarded using Relay.

Starting the system. Start in lst. 7 launches the SuS and monitoring system in tandem. The operation accepts the code signature g, as the entry point of the SuS, together with the map of executable monitors, Φ. As a safeguard that prevents the initial loss of trace events, the SuS is launched in a paused state (line 7.2) to permit the root tracer to start tracing the top-level system process. Root resumes the system (7.8), and begins its trace inspection in *direct* mode, as shown in line 7.10.

Message	Type (m.type)	Field name	Description
routed	rtd	rtr	PID p_t of the (ancestor) tracer that starts routing the message
		emb	The embedded trace event e or command c
detach command	dtc	tgt	PID p_s of the system process that is, from this point, traced by the new tracer
		iss	PID p_t of the new tracer issuing the detach command to the *router tracer*

Tbl. 5. Routed messages and detach command data field names

```
def START($g, \Phi$)
    # Pausing allows root tracer to be set
    # up; no initial message loss
    $p_s \leftarrow$ fork($g$) in paused mode
    $p_t \leftarrow$ fork(ROOT($p_s, \Phi$))
    return $\langle p_s, p_t \rangle$
end def

def ROOT($p_s, \Phi$)
    TRACE($p_s$, self())
    Resume system $p_s$
    $\sigma \leftarrow \langle \Pi \leftarrow \emptyset, \Phi, \Gamma \leftarrow \{\langle p_s, \circ \rangle\} \rangle$
    # Root tracer has no monitor
    LOOP$_\circ(\sigma, \bot)$
end def
```

Lst. 7. System starting operation and root tracer

B EXPERIMENT SET-UP AND EVALUATION

Inline monitoring implementation. We synthesise *automata-based* monitors from high-level correctness specifications. These monitors are encoded as executable functions that can be represented as an AST. Fig. 11 outlines how our monitors are inlined in the SuS. In step ①, the Erlang source code of the system is pared into the corresponding AST, step ② The Erlang compilation process contains a *parse transform* phase step ③ provides a hook that allows for the AST to be post-processed [Cesarini and Thompson 2009]. We leverage this mechanism through our custom-built weaver, step ④, that injects into the AST of the SuS the AST of the monitor in step ⑤. It performs two types of code transformations:

- C_1 *Monitor bootstrapping.* The function encoding the synthesised monitor is stored in the process dictionary (a key-value map) of the monitored system process to make it globally accessible from within said process;
- C_2 *Instrumentation points.* The AST of the system is instrumented with calls at the points of interest: these calls constitute the trace event actions that are to be analysed.

The instrumented calls in transformation C_2 retrieve the monitor function stored the process dictionary in transformation C_1, and apply it to the trace event in question. This function application on the event returns the *monitor continuation* that is used to replace the current monitor in the process dictionary. Our two-step weaving process produces the instrumented code in step ⑥ which can be subsequently compiled by the Erlang compiler into the application binary. We note that the same monitor ASTs synthesised for use in inline monitoring are used by our outline monitoring algorithm as well.

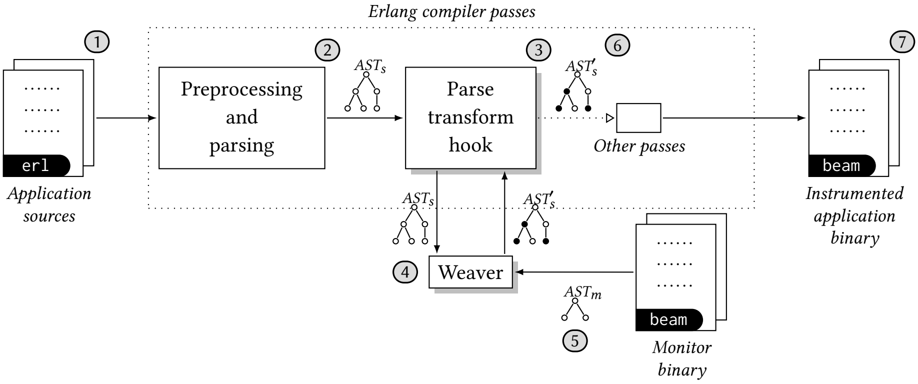


Fig. 11. Instrumentation pipeline for inline monitors

Validation of system model parameters. Our SuS, when configured with steady load, models web server traffic where the requests observed at the server are known to follow a Poisson process. The probability distribution of the RTT of web application requests is generally right-skewed, and can be approximated to a log-normal [Ciemiewicz 2001; Grove and Coddington 2005; Liu et al. 2001] or Erlang (a special case of gamma) distribution [Kayser 2017]. We conduct three experiments using steady loads fixed with $n = 10\text{k}$ and $w = 100$. $\Pr(send) = \Pr(recv)$ are varied through 0.1, 0.5 and 0.9 to establish whether the RTT for our chosen set-ups resembles the aforementioned probability distributions. Our results, summarised in fig. 12, were obtained as follows. The parameters for a series of candidate probability distributions (*e.g.* normal, log-normal, gamma, *etc.*) were estimated using Maximum Likelihood Estimation [Rossi 2018] on the RTT obtained from *each* experiment. We then performed goodness-of-fit tests on these parametrised distributions, selecting the most appropriate RTT fit for each of the three experiments. Our goodness-of-fit measure was derived using the Kolmogorov-Smirnov test. The fitted distributions in fig. 12 indicate that the RTT of our SuS confirms the findings reported in [Ciemiewicz 2001; Grove and Coddington 2005; Kayser 2017], which show that web response times follow log-normal or Erlang distributions.

Experiment Precautions. Further to the set-up parameters discussed in sec. 4.2, the following precautions were also taken:

- P_1 *Ten repeated readings.* The number of repeated readings to take was determined empirically based on the coefficient of variation, $\text{CV} = \frac{\sigma}{\bar{x}} \times 100$, that was calculated for experiments with different repetitions.
- P_2 $\Pr(send) = \Pr(recv) = 0.9$. Lower values of $\Pr(send)$ and $\Pr(recv)$ detract from the veracity of the experiments because slaves become frequently idle.
- P_3 *Scheduler utilisation.* Sampled every 500 ms asynchronously, not to affect the SuS. Samples were obtained using EVM function calls to get the most accurate reading. We did not measure the CPU at the OS-level, because the EVM keeps scheduler threads momentarily spinning to remain reactive, and this inflates the utilisation metric. This EVM feature could have been switched off, but we decided to use the default settings and instead, measure the utilisation internally.
- P_4 *Memory consumption.* Sampled every 500 ms asynchronously, not to affect the SuS. Samples were obtained using EVM function calls to get the most accurate reading.
- P_5 *Mean RTT.* Sampled every 10 % out of the total number of messages exchanged between master and each slave. The sampling window of 10 % was determined empirically via a series of tests. The RTT is calculated as a running mean of *each* sample taken; the overall drift w.r.t.

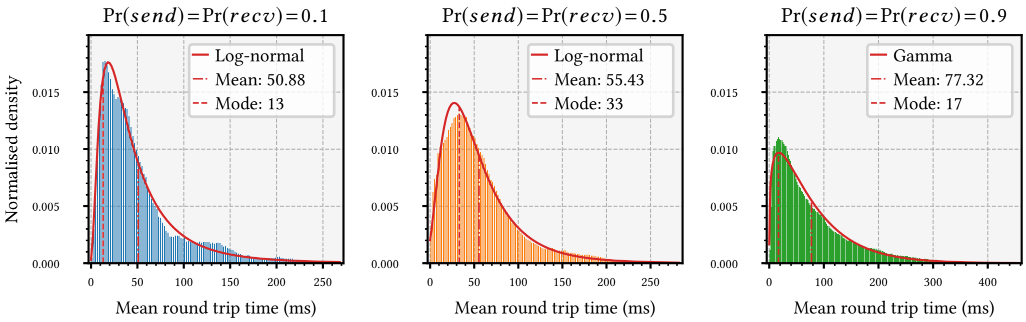


Fig. 12. Fitted probability distributions on mean RTT for steady loads of 10 k slaves

the mean calculated over all samples was ≈ ±1.4%. We gathered the RTT thus to avoid as much as possible perturbations in the SuS that would arise due to data collection.

- P_6 *Weighted mean.* We aggregated the sampling records collected from repetitions of the same experiment using the weighted mean to account for the differing number of records counts that were obtained at each run.
- P_7 *Randomisation seed.* We fixed the randomisation seed to ensure experiment repeatability.

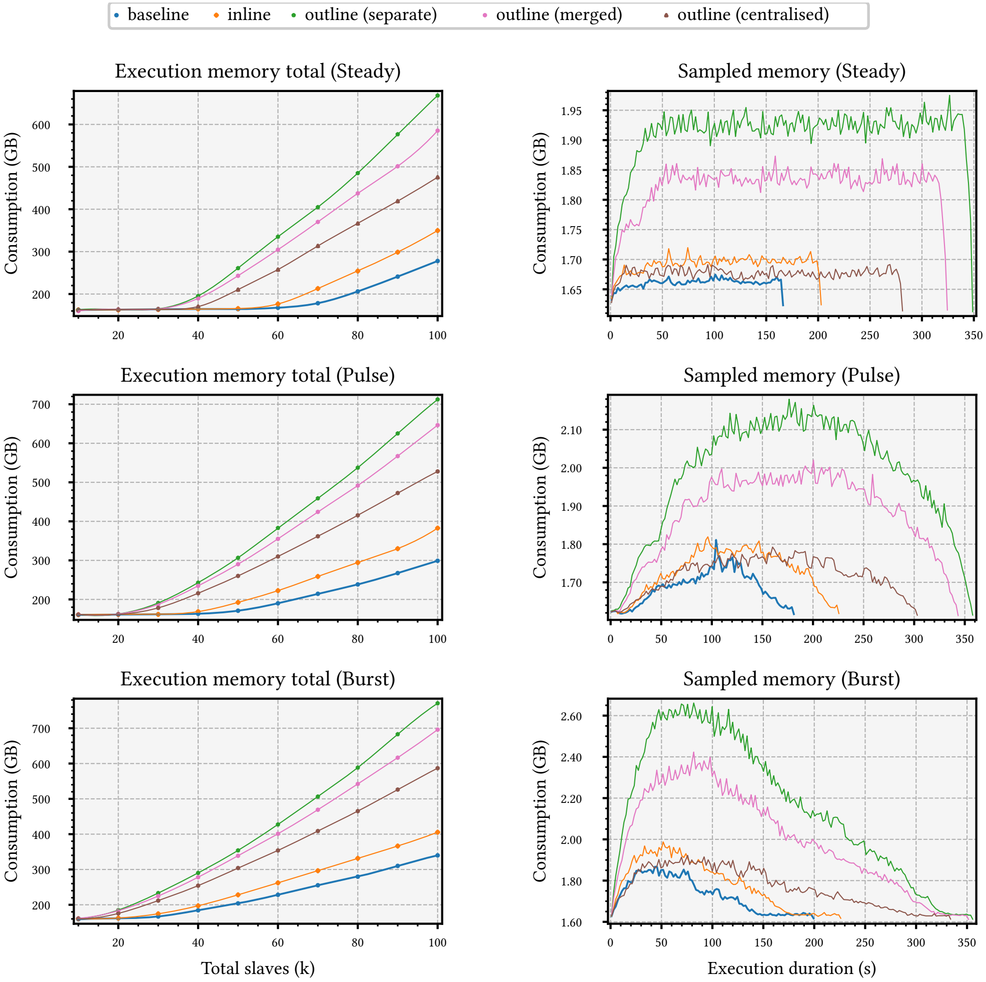


Fig. 13. Total and sampled memory consumption over entire execution duration with 100 k slaves

C SUPPORTING DATA PLOTS

The plots in figs. 14–17 have been fitted with linear, quadratic and cubic polynomials where the R^2 is above 0.96.

Total memory consumed. Fig. 13 shows the total memory consumed and sampled memory during the experiment runs conducted under Steady, Pulse and Burst loads for the case study with $n = 100$k slaves. Note that unlike in figs. 9 and 10, the y-axis is labelled in GB. The total memory consumed plotted on the left in fig. 13 corresponds to the *area* under the sampled memory plots on the right. Decentralised outline monitoring consumes the most memory, while the centralised version falls midway between decentralised outline (separate) and inline monitoring. The sampled memory plots reveal that centralised outline monitoring consumes less memory than inline monitoring on average, but does so for a longer time period. This is especially noticeable in the Steady and Pulse plots, suggesting the memory overhead in centralised outline monitoring is induced in a more consistent manner. Our sampled memory plots reflect the shapes of the loads applied, although these extend for a longer duration that goes beyond the original loading time of $t = 100\,\mathrm{s}$ (see fig. 6).

Moderate loads (fitted data plots).

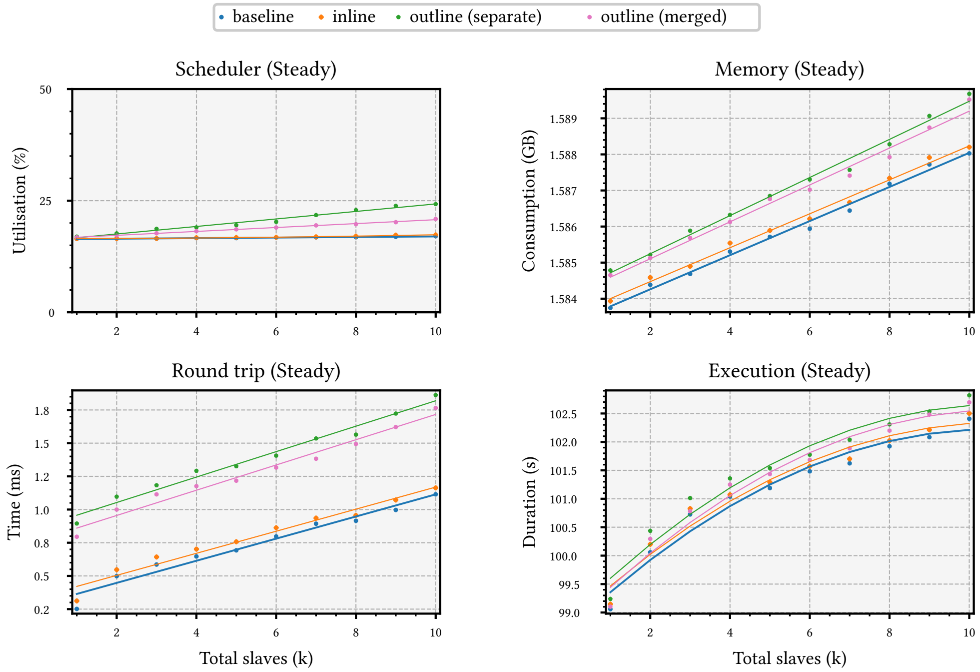


Fig. 14. Mean runtime overhead for monitoring the master and slave processes 10 k slaves

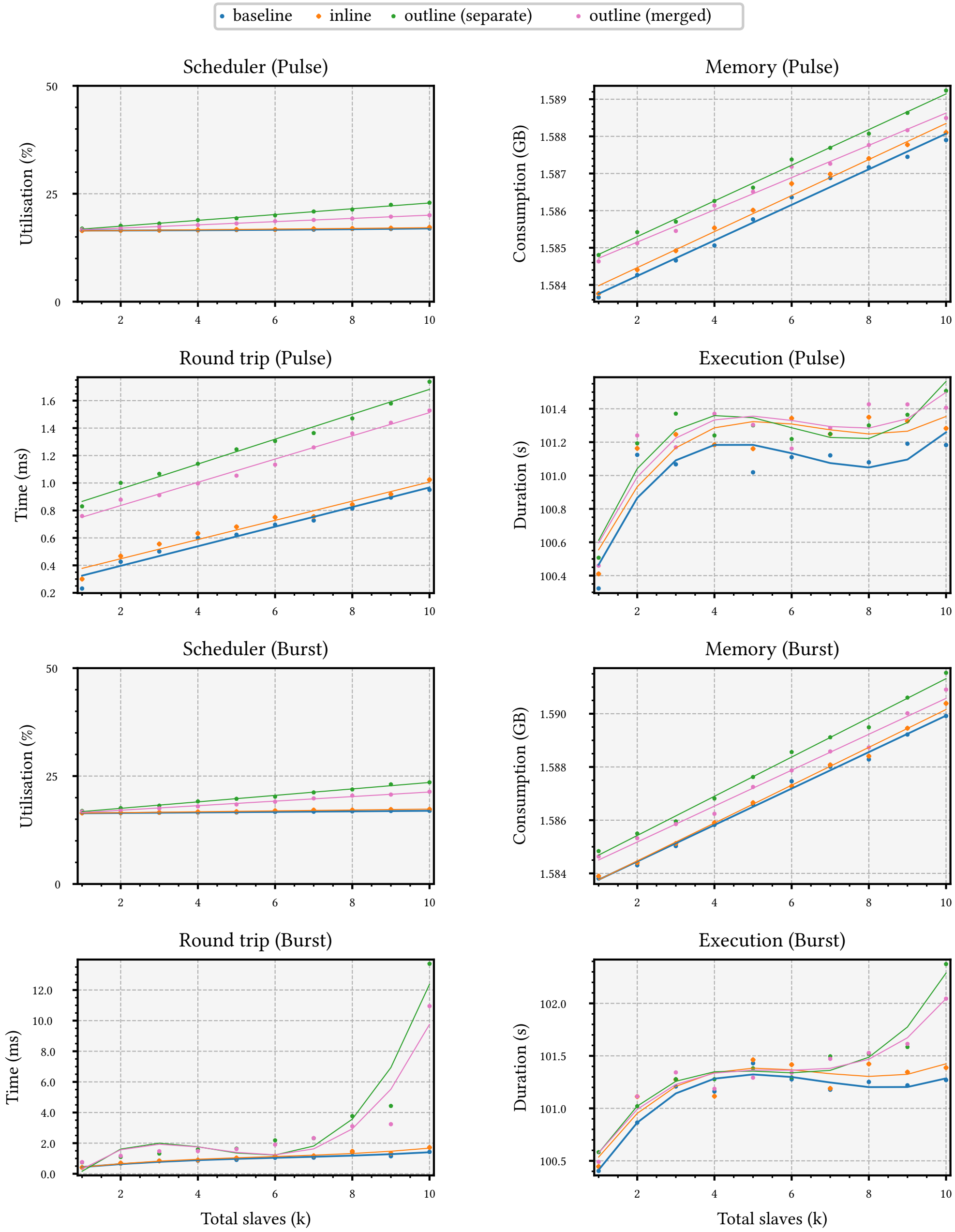


Fig. 15. Mean runtime overhead for monitoring the master and slave processes 10 k slaves (cont.)

High loads (fitted data plots).

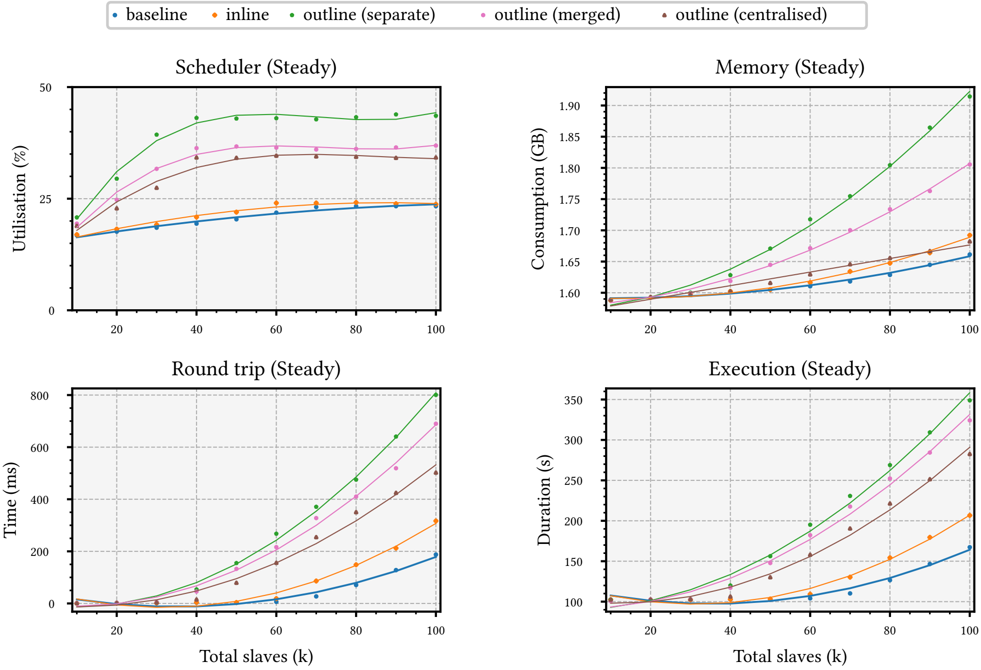


Fig. 16. Mean runtime overhead for monitoring the master and slave processes (100 k slaves)

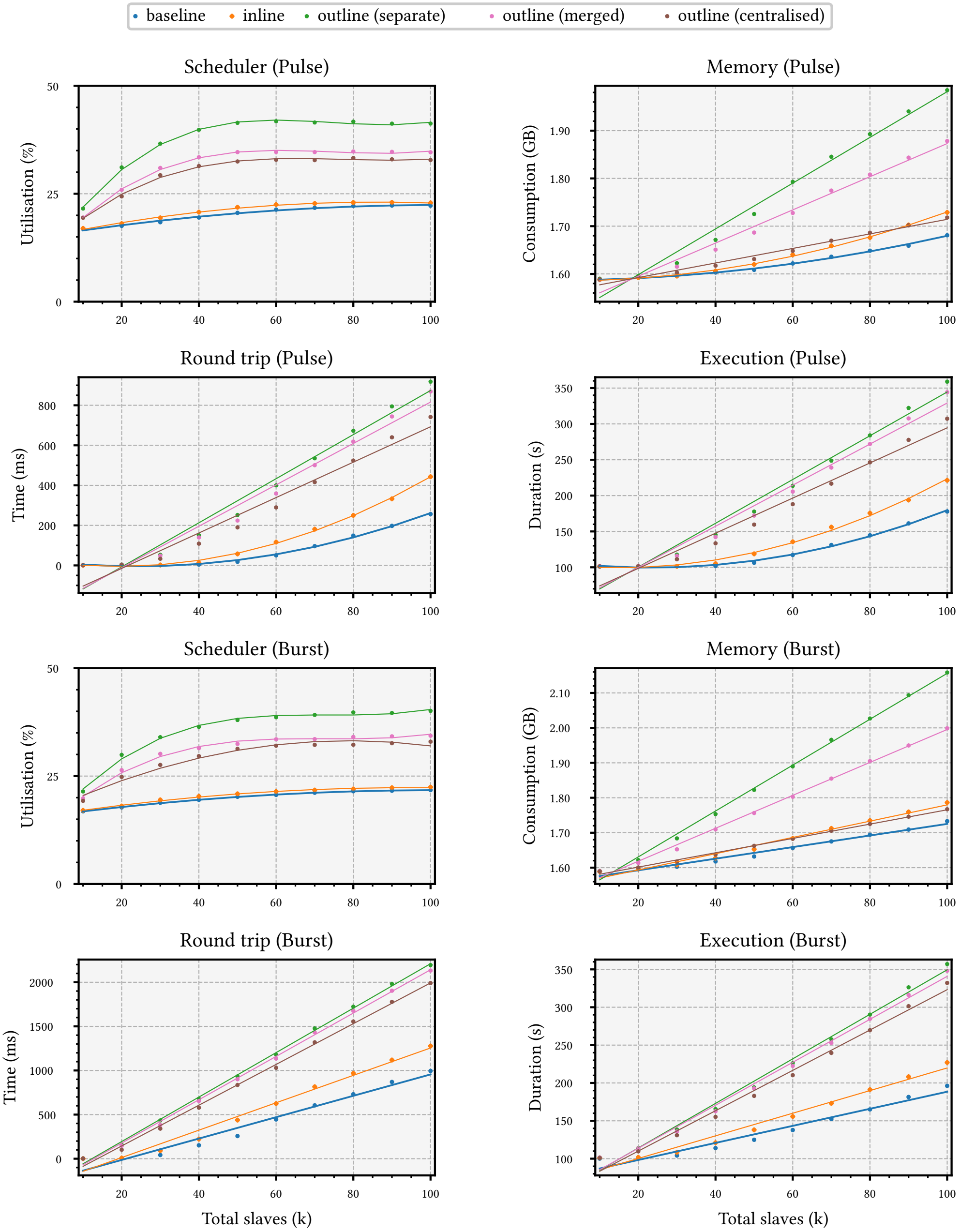


Fig. 17. Mean runtime overhead for monitoring the master and slave processes (100 k slaves, cont.)

Estimated overhead on slaves.

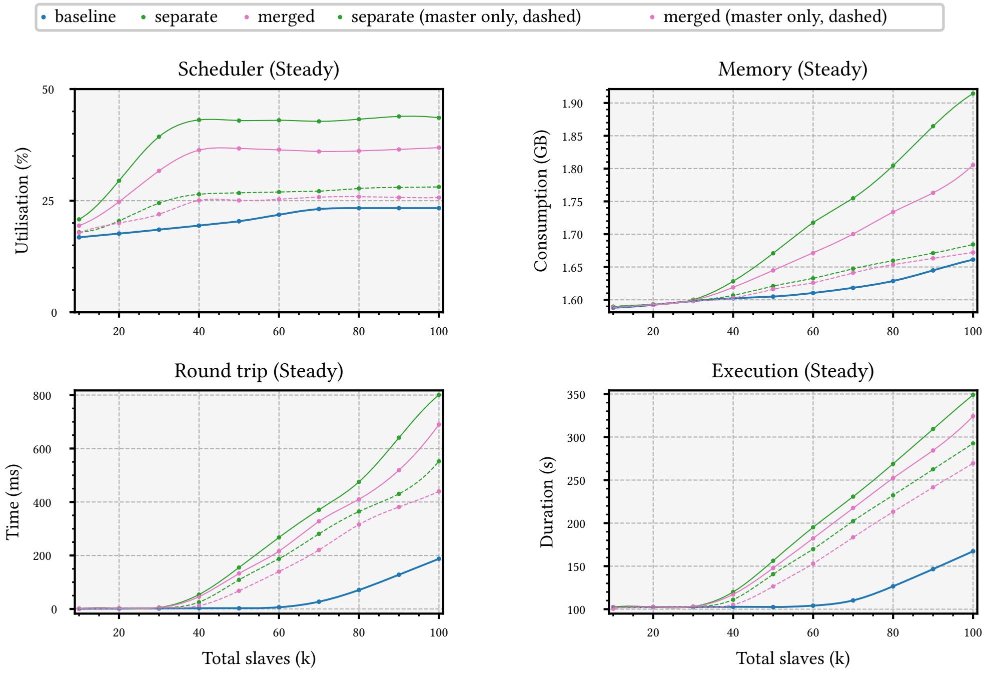


Fig. 18. Mean runtime overhead for monitoring the master process only (100 k slaves)

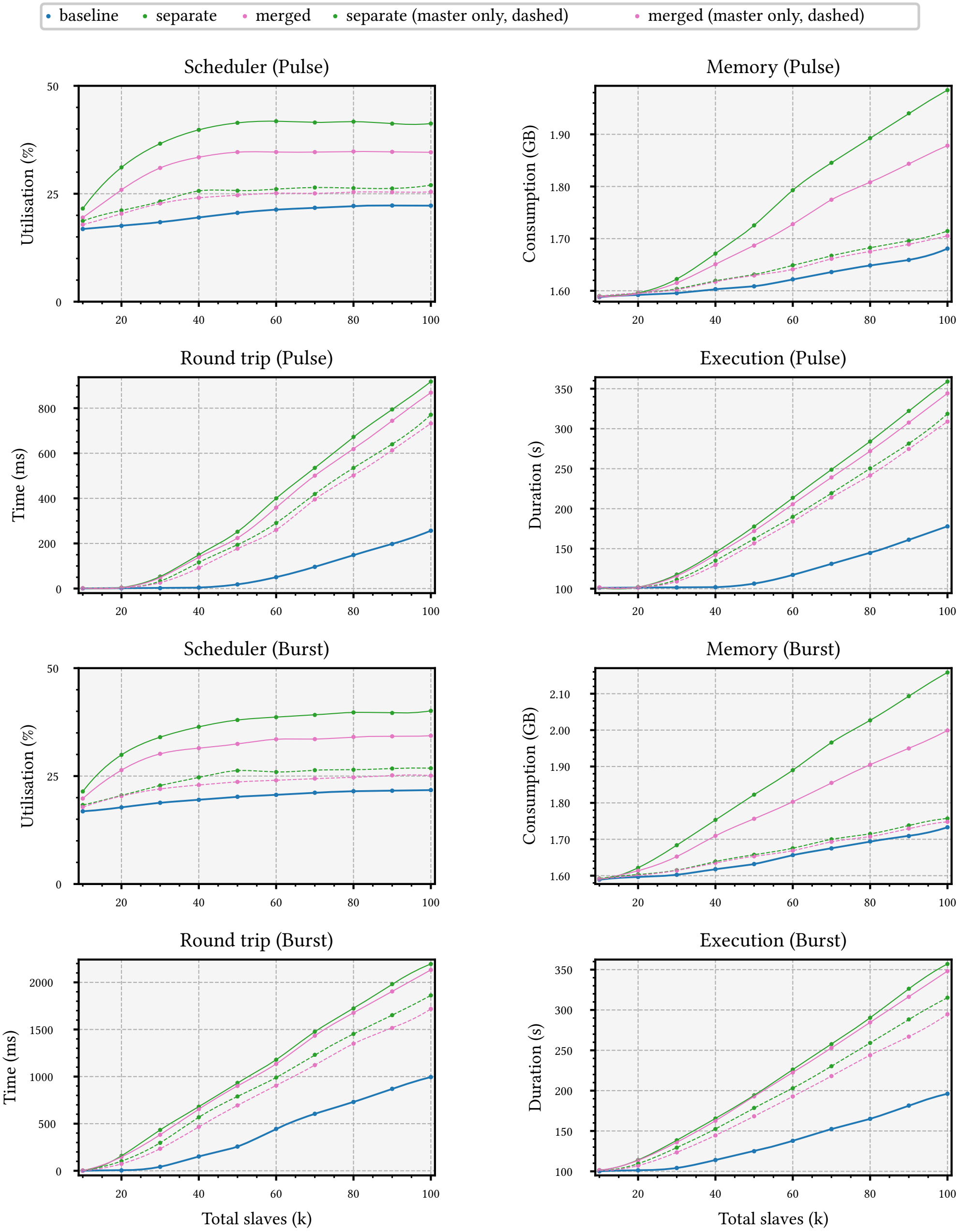


Fig. 19. Mean runtime overhead for monitoring the master process only (100 k slaves, cont.)