polyLarva: Runtime Verification with Configurable Resource-Aware Monitoring Boundaries

Christian Colombo[1], Adrian Francalanza[1], Ruth Mizzi[1], and Gordon J. Pace[1]

Department of Computer Science, University of Malta
{christian.colombo | adrian.francalanza | rmiz0015 | gordon.pace}@um.edu.mt

Abstract. Runtime verification techniques are increasingly being applied in industry as a lightweight formal approach to achieve added assurance of correctness at runtime. A key issue determining the adoption of these techniques is the overheads introduced by the runtime checks, affecting the performances of the monitored systems. Despite advancements in the development of optimisation techniques lowering these overheads, industrial settings such as online portals present new challenges, since they frequently involve the handling of high volume transaction throughputs and cannot afford substantial deterioration in the service they provide.
One approach to reduce overheads is the deployment of the verification computation on auxiliary computing resources, creating a boundary between the system and the verification code. This limits the use of system resources with resource intensive verification being carried out on the remote-side. However, under particular scenarios this approach may still not be ideal, as it may induce significant communication overheads. In this paper, we propose a framework which enables fine-tuning of the tradeoff between processing, memory and communication monitoring overheads, through the use of a user-configurable monitoring boundary. This approach has been implemented in the second generation of the Larva runtime verification tool, polyLarva.

1 Introduction

Due to its scalability and reliability, runtime verification [2] is becoming a prevalent technique for increasing the dependability of complex, security-critical, concurrent systems. Runtime verification broadly consists in checking for the correctness of the *current* system execution at *runtime*; it avoids checking alternative system execution paths and this, in turn, helps mitigate state-explosion problems associated with exhaustive techniques such as model checking. In scenarios where delayed correctness violation detection is unacceptable, runtime verification needs to be carried out in a *synchronous* fashion with the execution of the system. Invariably, synchrony increases the interaction with the system being monitored [1] and introduces overheads whose effects, particularly at peak times of system load, are hard to predict and may affect adversely the system behaviour.

[1] Although the monitoring for system events and the verification of the generated events against a specification are different operations, they are generally subsumed by the term *runtime verification* or *runtime monitoring* in the literature. We will use the terms interchangeably.

This issue is particularly relevant to scenarios such as online betting and e-commerce systems, where systems need to adequately handle multiple concurrent requests, resulting in uneven loads with peaks at particular times *e.g.*, during an important sporting event. The sheer size and complexity of such online portals, compounded with the security-intensive nature of their execution (carrying a significant probability of fraud attempts) makes synchronous runtime verification a good candidate for increased runtime assurance. However, for this to be viable, the overheads introduced by the monitoring should not compromise the availability of system resources at any stage of the execution of the system.

A possible approach to reduce the system overheads introduced by runtime verification is to deploy the synchronous verification processes onto *separate computing resources*: events of interest generated by the system are sent over to the remote-side (resources), where the necessary monitoring computation takes place and, as soon as the *remote-side* deduces that the system has not violated the specification, control is returned back to the *system-side* thereby allowing the system to proceed. This approach promises to be effective when monitoring highly parallel systems because the synchrony required between monitor and system usually concerns only a *subset* of the system processes. Stated otherwise, in a system with a high degree of parallelism, the processes that are not covered by the current runtime check may continue executing unfettered on the system-side, without being burdened by the cost of the verification computation. In cases of resource-intensive monitoring, this cost in (monitored) system performance offsets any additional slowdown stemming from the added communication overhead introduced by the distributed monitoring architecture.

For instance, consider an e-commerce system handling transactions of users that are categorised as either greylisted (untrusted) or whitelisted (trusted) and a correctness property for each greylisted user involving a computationally expensive statistical analysis of all financial transfers performed by that user. Performing the greylisted user checks remotely frees the system-side from the associated computational overhead, ensuring that whitelisted users (and other greylisted users not performing a transfer at the moment) are not affected by the monitoring computation.

However, shifting all monitor checking to the remote-side is no silver bullet. In certain cases, performing verification checks remotely is impractical because these checks would require access to resources and state information kept by the system *e.g.*, a local database; a remote evaluation of these verification conditions would require either resource replication or expensive remote access of resources. Furthermore, even when verification checks do not require access to system-side resources, there are still instances where shifting all runtime verification checks to the remote-side does not yield the lowest level of overhead. For instance, whenever the slowdown associated with communication outweighs the benefits gained from shifting monitoring checks to the remote-side, it is advantageous to perform the verification check at the system-side, circumventing any communication overhead. In the above e-commerce example, we may have a property stating that greylisted users may not transfer more than $1000 in a single transaction: since the cost of a single integer comparison is typically less than that of communicating with the remote-side, it may pay to monitor the property on the system-side.

Even more complex situations may also arise. Consider again the e-commerce system which checks greylisted users for fraud. In this case, in order to minimise monitoring overhead, it may be advantageous to *split* the verification check across the system-side and remote-side. More precisely, in order to avoid communication overheads the check of whether a user is greylisted or not is performed on the system-side; this obviates the need for any communication with the remote-side (where the expensive statistical check is performed) whenever the user is whitelisted.

These examples attest that, in settings where verification can be done remotely, adequate control over where and when the verification computation is executed is essential for minimising the overheads of runtime monitoring security-critical concurrent systems. In spite of this need, the existing technologies that can be used for remote-verification (*e.g.*, [8]) do not offer structuring mechanisms to support the fine-grained distribution control just discussed. In this paper, we propose a runtime verification system with a *configurable monitoring-boundary* enabling the user to decide which verification tasks are to be computed on the system side and which are executed on the remote side. This approach has been implemented in the tool polyLarva, the second generation of the tool Larva [5], where language-support is provided to enable the user to easily stipulate the system-monitor boundary. Such added flexibility empowers the user to decide the best allocation strategy for the runtime check at hand.
The contributions of the paper are:

1. the presentation of a framework enabling the fine tuning of system-side and remote-side computations;
2. showing the feasibility of the framework through its realisation in the polyLarva tool; and
3. evaluating the approach through a number of case studies, measuring the effect of changing the configuration of the monitoring boundary.

The paper is organised as follows. In Section 2 we present our specification logic, the target architecture and the mapping from the logic to this architecture. Section 3 introduces the case study used to demonstrate the utility of our approach and Section 4 discusses the tests performed and the results obtained. Section 5 reviews related work and Section 6 concludes.

2 Configuring the Monitoring Boundary

We limit ourselves to an interpretation of runtime verification that consists of two main components. The behaviour of the system being monitored is characterised by a stream of *events* which are analysed by a *monitor* which may be composed of various monitoring tasks such as condition valuations and state updates. Our proposed framework provides mechanisms, in the guise of a monitoring boundary, for controlling the localisation of the monitoring tasks, when setting up the runtime verification configuration. One can therefore use this boundary to minimise the execution overhead introduced by the instrumented runtime verification on the system. An important caveat is that the partitioning of monitoring components between the system-side and remote-side largely depends on the kind of logic used and the forms of actions handled by the monitor.

For instance, if the logic is a simple one limited only to identifying undesirable events, *i.e.,* no temporal event ordering, then verification essentially requires little computation, which does not leave much scope for a monitoring boundary. However in more complex logics, such as LTL *e.g.,* [7, 10], the monitor has to keep track of how much of the LTL formula one has already matched. Similarly, in logics expressed as symbolic automata *e.g.,* [5], one can directly program the monitor state; this gives more flexibility as to how much and which parts of the code are to be executed on either side. Analogously, the extent and the nature of the actions taken by the monitor also determines the level of placement manouvering that can be performed to minimise overheads.

2.1 Monitoring using polyLarva

Our monitoring framework polyLarva uses a guarded-command style specification language. Properties are expressed as a list of rules of the following form:

$$event \ \mid \ condition \ \mapsto \ action$$

Whenever an *event* (possibly carrying parameters) is generated by the system, the list of monitor rules is scanned for rule matches relating to that event. If a match is found, the expression specified in the *condition* of the rule is evaluated and, if satisfied, the *action* is triggered.

Example 1. Consider a scenario in which one desires to monitor whether a greylisted user pays using a credit card after verifying it, blacklisting offending users if this rule is violated. This may be expressed in terms of the following two rules for each active user:[2]

$$\begin{array}{ll} register(user, card) & \mid isGreylisted(user) \mapsto registeredCards[card] := true; \\ pay(user, card) & \mid isGreylisted(user) \wedge \neg registeredCards[card] \mapsto blacklist(user); \end{array}$$

In the above rules, *register* and *pay* are system events, parametrised by the values *user* and *card*; $isGreylisted(user)$ and $\neg registeredCards[card]$ are conditions, while $registeredCards[card] := true$ and $blacklist(user)$ are actions taken by the system. While some conditions and actions may be cheap to perform (e.g. $registeredCards[card] := true$ consists of a single assignment), others may be more computationally expensive (e.g. a condition checking whether an ongoing transaction is fraudulent or not may require more computation).

2.2 The target architecture

Our approach gives sufficient high-level mechanisms such that a user can configure the monitoring boundary for synthesised lists of rules discussed in Section 2.1. This architecture allows for the possibility of having two sub-monitors that are automatically synthesised from a polyLarva script, one on the system-side and the other on the remote-side as shown in Figure 1. Communication across nodes is carried out using socket connections: socket-based communication is low-level enough to be optimisable, while also providing the flexibility of technology-agnosticism.[3]

[2] For simplicity, we assume that the array of registered cards starts off with all cards unregistered, and that only a single user may access a particular card.

[3] As future work polyLarvawill be extended to support multiple technologies.

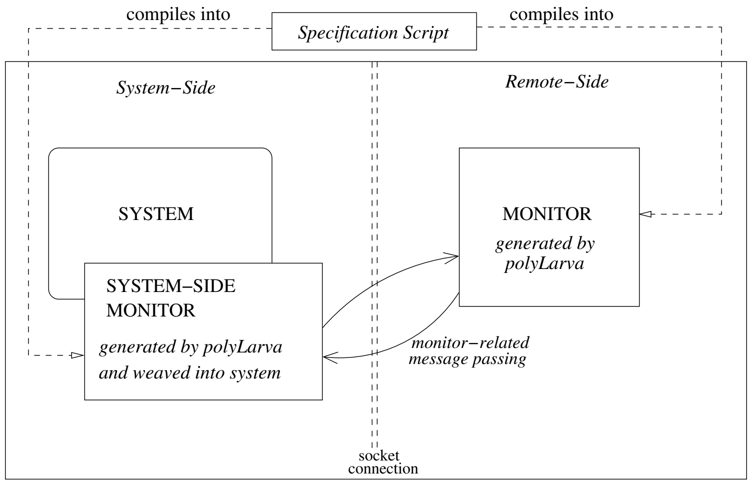

Fig. 1. Separation of monitor and system in polyLARVA

The typical monitoring control flow over such an architecture proceeds as follows: (i) Events generated on the system-side trigger matching rules; (ii) Once a matching rule is found, the condition to be evaluated is determined. This condition is made up of a conjunctive sequence of (arbitrarily complex) basic conditions that are joined using normal negation, disjunction and conjunction operators. The basic conditions are categorised as either *system-side conditions*, *i.e.*, predicates which are to be evaluated on the system-side, or *remote-side conditions*, *i.e.*, predicates which should be evaluated at the remote-side; (iii) The evaluation of the condition starts and control is passed back and forth between the monitoring component on the system-side, whose role is that of evaluating the system conditions, and the monitoring component on the remote-side, whose role is to evaluate monitor conditions and coordinate control associated with the boolean operators. (iv) If the condition succeeds, the list of actions dictated by the matched rule is executed sequentially. Once again, actions are categorised as system- or remote-side, which entails passing control back and forth between the two sides; (v) Upon termination of the last action, control is passed back to the system to proceed.

Numerous variations and optimisations can be performed over the basic architecture of the above control flow. For instance, it may be beneficial to perform the rule matching process on the system-side, which avoids the overhead of communicating the event to the remote-side when a rule is not matched. In cases when the matched rule consists solely of system conditions and system actions, communication with the remote-side can be avoided altogether if the system-side monitoring component is entrusted with rule coordination capabilities. A similar situation may arise if part of the monitoring state is held at the system-side. Most of these optimisations are however case-dependent and do not apply to all verification specifications in general. Thus, such decisions cannot be automated: their control is best elevated at specification level and left to the verification engineer.

2.3 The Implementation

In polyLarva the rules presented in Example 1 are coded as shown in Program 2.1. The rules are enclosed within a `rules` block, ①, which replicates them for every new user session that is opened, ②. The `state`, `conditions` and `actions` blocks define specification-specific monitor state, conditions and actions respectively that are used later in the rules section.

Program 2.1 Monitoring greylisted users for card registration

```
② upon (newUserSession(u))  {
    state {
      remoteSide  { boolean[] registeredCards;  }
    }
    conditions {
      systemSide  { isGreylisted(u) = ... }
      remoteSide  { isRegistered(c) = { registeredCards[c] } }
    }
    actions {
      systemSide  { blacklistUser(u) = ... }
      remoteSide  { registerCard(c) = { registeredCards[c] := true } }
    }
    ① rules {
      register(u,c) \ isGreylisted(u) -> registerCard(c);
      pay(u,c) \ isGreylisted(u) && !isRegistered(c) -> blacklistUser(u);
    }
  }
```

In this example, `blacklistUser(u)` is a system action changing the internal state of the system relating to user `u`, whereas `registerCard(c)` is a remote action affecting the remote state `registeredCards` (a boolean array keeping track of which card numbers have been registered). Similarly, `isGreylisted(u)` is a system condition whereas `isRegistered(c)` is a remote condition, evaluated using the remote state `registeredCards`. As a result, the first rule's condition depends solely on system information, while the second refers to both the system state (whether a user is greylisted) and the remote state (whether the card was previously registered).

Deciding whether to place conditions and actions on the system-side or the remote-side to yield the minimum overhead is not straightforward and case-dependent. In the next section we use a real-life case study to investigate the alternatives.

3 Case Study

polyLarva has been used to monitor JadaSite[4] — an open source e-commerce solution written in Java, offering a range of features from back-office administration to au-

[4] http://www.jadasite.com

tomatic inventory control and online sales management. The case study focusses on adding monitoring functionality to analyse user transactions in order to detect and prevent fraudulent transactions. Performing fraud detection in an offline manner, through the analysis of logs, is not ideal since by the time the analysis is carried out, a fraudulent user may have affected multiple transactions. Synchronous monitoring, enabling blocking a user upon suspicion of fraud, is desirable. Unfortunately, effective fraud detection typically requires costly analysis of the user's history, involving multiple database accesses and processor-intensive calculations. Furthermore, since at peak times the system may have multiple users performing transactions concurrently, reducing the overhead is crucial.

The most straightforward partitioning is to place only parts which refer to the system state on the system-side, and placing the rest of the code on the remote-side. Through the code in Program 2.1 we see how polyLarva allows the actions and conditions to be tagged in such a manner (highlighted in grey) instructing this code partitioning. However, partitioning of code can involve more intricate situations.

Example 2. Consider the following extension to Example 1 with the property that untrusted customers are not allowed to perform a payment if the probability of a fraud being committed is above a certain threshold.

$$pay(user, card) \mid \neg isWhitelisted \wedge isFraudulent(user) \mapsto failTransaction(user, card);$$

The check for possible fraud is assumed to be a computationally expensive statistical analysis, while the decision of whether a customer is whitelisted is assumed to depend on the number of safely concluded payment transactions the user has already performed. The monitoring execution of this property is depicted in Figure 2[left]), where the monitor is notified of the relevant events (*newUserSession* and *pay*), updates its customer state (increasing the number of transactions), checks any other appropriate conditions (checks the transaction with the customer history for fraud patterns), and performs actions accordingly (stopping the transaction) before returning control to the system. Program 3.1 shows an encoding of this example in polyLarva, except for the location of state, conditions and actions, which will be discussed later.

Choosing the location of the monitoring code depends on different issues. For instance, since the fraud check ③ is assumed to be a resource-intensive operation, locating it on the remote-side relieves the system of the overhead, thus remaining responsive to the rest of the users. The stopping of a transaction ⑤ is an action which affects the system, meaning that it has to be located there. Finally, the transaction count is kept as a monitor state, ①, read by the monitor condition `isWhitelisted` ③ and written to by the verifier action `incrementTransactionCount` ⑥. This suggests that the three entities should be co-located so as to reduce additional communication for remote state access. Since the computation associated with this state is lightweight, it can be feasibly located on the system-side without affecting the system performance in any considerable manner. The script which specifies such a boundary configuration is shown in Program 3.2, with the resulting communication pattern shown in Figure 2[right].

Note that control may have to go back and forth the two sides multiple times due to the way the rules are structured. From this, it should be clear that (*i*) from a communication point of view, it would appear to be desirable to commute the two conjuncts on

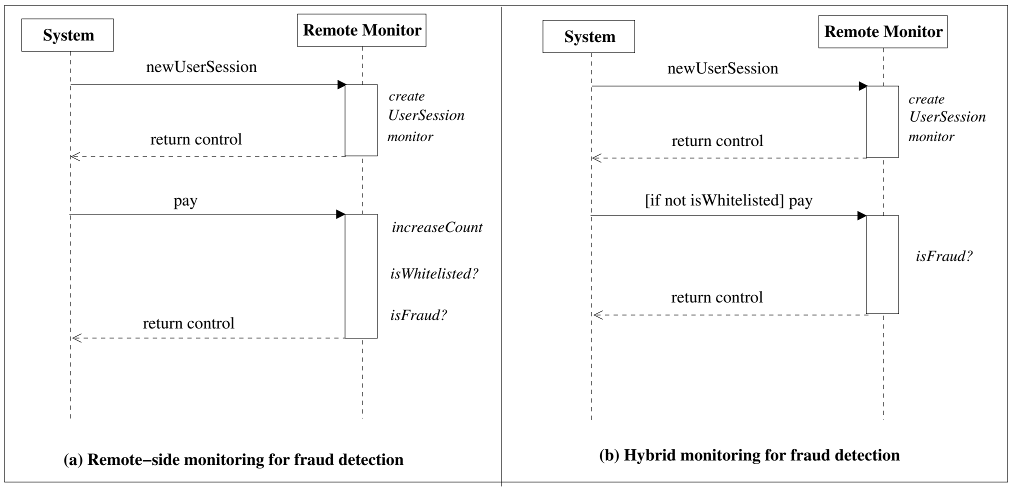

Fig. 2. Monitoring for fraud detection.

Program 3.1 Monitoring for fraud detection

```
upon { newUserSession(u) } {
  state {
    ① integer transactionCount;
  }
  conditions {
    ② isWhitelisted = ...transactionCount...
    ③ isFraudulent(u) = ...CPU intensive algorithm...
  }
  actions {
    ④ incrementTransactionCount = transactionCount++;
    ⑤ stopTransaction(u,c) = ...
  }
  rules {
    ⑥ startTransaction \ true -> incrementTransactionCount;
    ⑦ pay(u,c) \ !isWhitelisted && isFraudulent(u) -> stopTransaction(u,c);
  }
}
```

Program 3.2 Tagging the case study with a monitoring boundary

```
② upon  newUserSession(u)  {
  conditions {
     systemSide  {
       isWhitelisted(u) = {
         EntityManager em = JpaConnection...
         ... query = em.createQuery(sql);
         Long custTransNo = (Long) query.getSingleResult();
         return (...custTransNo.intValue()...)
       }
     }
     remoteSide  { isFraudulent(u) = ...CPU intensive algorithm... }
  }
  actions {
    systemSide  { failTransaction(u,c) = ... }
  }
  ① rules {
    pay(u,c)  !isWhitelisted(u) && isFraudulent(u) -> failTransaction(u,c);
  }
```

line ⑦ so as to avoid control going from the monitor to the system side and back, but the high computational cost of checking for fraudulence means that we would prefer to start by performing the cheaper check for user trust first; (*ii*) If the system already keeps a record of payment transaction counts per user in its database, one may locate the trust checking condition to the system node, thus reducing communication by removing rule ⑥. In general, deciding the monitoring boundary can be seen as a minimisation problem having: a system-side and a remote-side, a number of (weighted) monitoring tasks, and a number of weighted communication signals as shown in Figure 3 (with task represented as boxes and communication as arrows). A minimal placement is one with the least number weighted boxes at the system-side, as few communication weights as possible, and having conditions which fail with a high probability as early as possible.

In the next section we give case study results corresponding to different monitoring boundary configurations showing how selected configurations can contribute to a non-negligible reduction of the monitoring overhead in a real-life case study.

4 Results

We have carried out a series of empirical tests showing how different monitoring configurations have a substantial impact on the performance of a monitored system. The noticeable overhead differences justify the need for a verification technique that permits flexibility with respect to the instrumented monitoring configurations *i.e.*, a configurable monitoring boundary. The tests also show how, in practice, the impact on system performance cannot always be fully predicted at instrumentation time. Thus, a level of abstraction that gives high-level control over the monitoring boundary, such as that

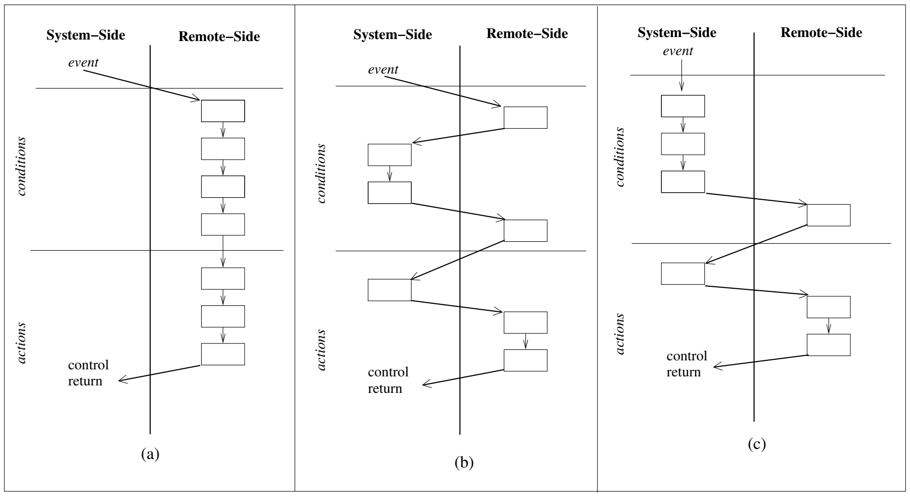

Fig. 3. Monitor placement

presented for polyLarva, is required to facilitate the re-configuration of instrumented monitors.

Our tests are carried out on the JadaSite ecommerce system introduced in Section 3. Since the application we consider consists in a web portal handling an extensive amount of concurrent user request, an important aspect of system performance affected by monitor overheads is the Average time taken for a Payment Transaction (APT) to be processed, which directly translates to system responsiveness. Our tests synthesise runtime monitors for different monitoring boundary configurations for Program 3.1 using polyLarva. Three distinct monitoring configurations were considered for our tests:

System-side monitoring (SM): Verification is entirely deployed on the system-side, running on the same address space as the system (all tags are *systemSide*).

Remote-side monitoring (RM): Verification checking exclusively is carried out on the remote-side, running on a separate machine from the system (all tags are *remoteSide*).

Hybrid monitoring (HM): Verification checking is split between the system-side and the remote-side as in Program 3.2.

For each user-load level, we also benchmark the performance of the Unmonitored System (US), which helps us calculate the overhead introduced by each monitoring configuration.

The performance of the monitored system is benchmarked subject to user loads ranging from 40 to 120 concurrent user requests, involving operations such as adding items to the shopping cart, confirming payment details and executing the payment transaction. Load testing of the JadaSite web application, in conjunction with runtime monitoring, is carried out using Apache JMeter 2.5.1[5]. Java SE 1.6 is used for the compila-

[5] http://jmeter.apache.org/

tion of the JadaSite system source code and for the generation and compilation of the polyLarva monitors. JadaSite is deployed on an Apache Tomcat 7.0.23[6] server running on an AMD Athlon 64 X2 Dual Core Processor 6000+ PC, 4GB RAM, running Microsoft Windows 7. The remote-side consisted of a separate machine having an Intel(R) Core(TM)2 Duo CPU T6400, 2GB RAM with Microsoft Windows 7 operating system.

An important aspect affecting our tests is the ratio between whitelisted and greylisted users. This is because the verification checks specified in tests such as Programs 3.1 differentiate between whitelisted users and greylisted ones: greylisted users are subject to a monitoring condition requiring a computationally expensive fraud check whereas whitelisted users are not. For our experiments, two-thirds of the users are chosen to be whitelisted and the rest are considered to be greylisted. This ratio reflects more of a realistic deployment of the system where most of the users are regular users; the majority of these regular users are most likely to become whitelisted (trusted) after a probation period during which their transactions do not violate any verification checks.

Table 1. Average payment transaction duration for each user *wrt.* user load (in secs)

Setup	40	50	60	70	80	100	120
US	11.4	17.7	24.0	28.7	37.4	56.2	69.2
SM	15.5 (35%)	21.5 (21%)	28.3 (17%)	34.6 (20%)	43.1 (15%)	67.8 (21%)	107.2 (55%)
RM	13.5 (18%)	19.7 (11%)	26.8 (11%)	34.4 (20%)	41.9 (12%)	60.4 (8%)	89.9 (30%)
HM	14.2 (24%)	22.7 (28%)	26.1 (8%)	30.9 (8%)	39.2 (5%)	58.4 (4%)	84.0 (21%)

Table 2. CPU processing units used *wrt.* to user load

Setup	40	50	60	70	80	100	120
US	16598	21836	28340	34026	39416	58097	73266
SM	17591 (6%)	22991 (5%)	31315 (10%)	36886 (8%)	43295 (10%)	63275 (9%)	103850 (41.7%)
RM	15215 (-8%)	20971 (4%)	26470 (-6%)	33729 (-1%)	42353 (7%)	58981 (1%)	93792 (28%)
HM	16187 (-2%)	23381 (7%)	28333 (0%)	32199 (-5%)	41195 (4.5%)	60636 (4.4%)	86216 (17.7%)

The main results of the experiments measuring the APT and the respective CPU usage at the system-side under different configurations can be found in Table 1 (depicted in Figure 4) and Table 2. When compared to the base APT of the unmonitored system in Table 1, it becomes evident that system-side monitoring (SM row) introduces substantial overheads, peaking at a level of 55% increase in APT when the tests hits a user load of 120 concurrent transations. Table 2 indicates that the sharp increase in APT can be attributed to the increase in CPU usage at the system-side, depleting resources from

[6] http://tomcat.apache.org/

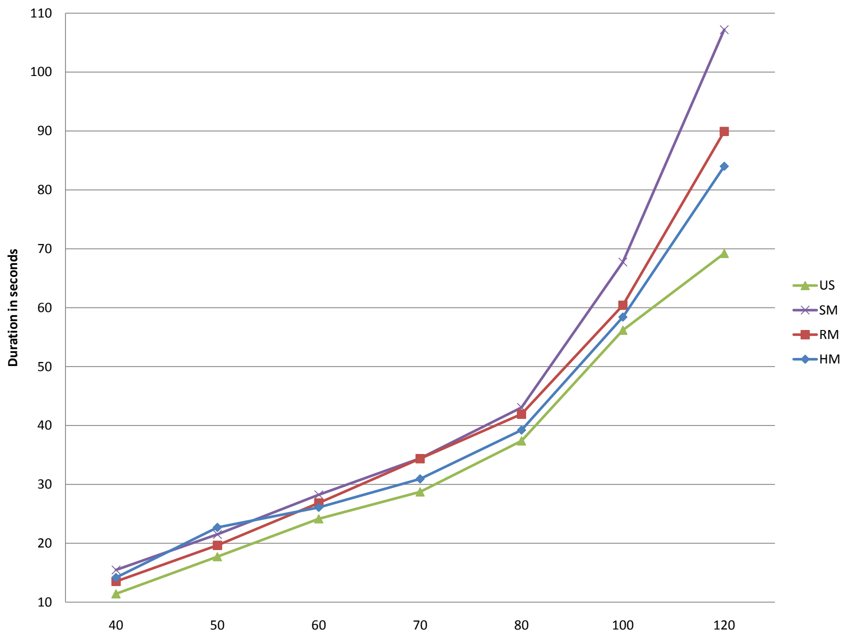

Fig. 4. The average payment processing time *wrt.* user load

the execution of the system.[7] Such a deterioration in system responsiveness will most likely discourage the adoption of runtime verification checks over the live system.

Table 1 shows that one effective way of substantially reducing overhead is by employing additional resources at the remote-side and shift all verification to the auxiliary side (RM). Figures however show that, at a load of 120 users, overhead spike even with this monitoring strategy reaching a level of 30% overhead increase. One possible explanation for this is that the communication channel between the system-side and remote-side becomes saturated causing a bottle-neck in the verification operations.

Our proposal towards solving this problem is to have a Hybrid setup, HR, leaveraging parts of the verification on the system-side. Figures in Table 1 show that at low user request loads, *e.g.*, 40 and 50 users, RM performs better than a hybrid approach because more of a less-scarce resource *i.e.*, the communication channel, is being used as opposed to CPU usage at the system-side. However, at higher user loads such, *e.g.*, 80, 100 and 120 users, the balance tips in favour of shifting some verification on the system-side, *i.e.*, HR, where the overheads are consistently less that in the case of RM; at these levels, a hybrid approach manages to approximately *half* the overheads introduced by an extreme remote-side monitoring strategy. In more realistic distributions where the level of untrusted (greylisted) users is even lower, a hybrid approach yielded even better results. We conducted further experiments (see Table 3) where despite user

[7] Fraud checking was not memory intensive and, as a result, memory usage was not an issue.

load increases, the number of greylisted users was fixed at 14 users. The results yield more significant gains as the number of users increase.

Table 3. APT per number of users (in secs) with increasing whitelisted users

Setup	40	70	100	120
US	11.4	28.7	56.2	69.2
SM	15.5 (35%)	35.2 (22%)	62.0 (9%)	99.2 (43%)
RM	13.5 (18%)	39.0 (36%)	60.9 (8%)	72.3 (4%)
HM	14.2 (24%)	34.3 (19%)	56.6 (1%)	69.6 (1%)

In conclusion, the CPU figures obtained in Table 2 for RM and HM at low user-request loads deserve further comment, since they appear to suggest that introducing monitors sometimes actually reduces CPU usage. This might be attributed to reduced context switching due to the blocked users waiting for monitor feedback.

One aspect which is hidden in this quantitative analysis is the fact that the hybrid approach allowed for the localisation of code which goes more naturally on the system side *e.g.*, code accessing data that is already computed on the system side. By contrast, in the remote monitoring approach one would have no option but to duplicate this computation and the associated data, introducing computation redundancy and additional space overheads. In fact, in cases where resources replication is either not feasible or undesirable,[8] a hybrid approach turns out to be the only viable solution between these two alternatives.

5 Related Work

Optimisation techniques for synchronous monitoring is a key issue in runtime verification. These techniques broadly fall under two main categories: event sampling techniques and static/dynamic analysis. In the first category [1, 6] only a subset of the system events generated are checked by the monitor, typically in line with some periodic overhead upper limit; this arrangement allows the verification instrumentation to give certain guarantees with respect to monitor overheads, at the expense of monitoring precision. In the second category, static analysis is performed on the monitored properties and their instrumentation [4, 3] in order to optimise their footprint. The first class of techniques are not directly applicable to the security-critical systems discussed in the Introduction since certain violations may go undetected. However a degenerate case of sampling may be used in real world instantiations of our approach acting as a method of last resort when the verification overheads overburden the system. The second class of techniques are complementary to our approach since the enhanced control over where to instrument monitors gives further scope for static analysis to optimise such placements.

[8] Issues such as data privacy may prove to be one such stumbling block.

To the best of our knowledge, Java-MaC [8] is the only runtime verification tool that implicitely places a boundary between the system and the verifier (albeit with no support for flexibility), by distributing verification across nodes and potentially lowering monitoring overheads. The monitor can however only be located on the verifier side; we argued earlier why this placement strategy may not always yield an optimal level of overheads. Other tools such as JavaMOP [9] and Larva [5] can support our proposed architecture *indirectly*, since they allow full Java expressivity for monitoring checks and actions. This permits monitor instrumentation to use monitoring actions to open connections and instruct remote deployment of verification checks. However, such an arrangement is far from ideal as it complicates immensely the specification of properties: it requires additional knowledge of Java distribution mechanisms, thereby discouraging the adoption of our proposed architecture. Moreover, this indirect approach clutters the monitoring code which, in turn, makes monitoring more error prone.

6 Conclusions and Future Work

We have proposed a novel runtime verification technique facilitating the engineering of runtime monitoring over highly parallel systems, thus minimising the inherent overheads introduced by the verification process. By elevating a configurable monitoring boundary to the specification level, the technique allows the user to offload computationally expensive verification checks to a remote site while leaveraging the added communication overhead by keeping lightweight verification checks at the site where the monitored system is executing.

The technique has been implemented as part of a runtime monitoring tool called polyLarva, which takes guarded-command style specification scripts and automatically synthesises the system instrumentation together with the respective monitor verifying the script. The tool allows the user to specify the location of where conditions and actions are to be executed; these delineations correspond to configurable monitoring boundary of the technique and give control over how system resources are managed.

We have also shown how our approach enables alternative monitor configurations that lower overheads through tests performed on an online portal handling multiple user requests in parallel. Our results indicate that different overhead savings can be obtained under different monitoring boundary specifications, depending on the level of service load experienced by the system and on whether monitoring is processing or communication intensive. These results show that while resource-intensive monitoring benefits substantially from remote monitoring, communication does not scale up as well as other resources. Balancing resources overheads against the communication incurred turned out to be essential to lower these overheads, and polyLarva facilitated the necessary fine-tuning immensely.

Future Work: We plan to extend our work in various ways. We intend to further our tests to deployment architectures involving more than one node for the remote-side of the monitoring boundary. We are also exploring ways how to integrate our optimisation technique with complementary techniques such as sampling. These efforts should yield even lower monitoring overheads which would increase the appeal of the technique to real-world scenarios with more stringent performance requirements, as opposed to security-critical systems.

Work is already underway to extend polyLarva so that it can handle monitoring of systems that are developed using *different technologies and languages*, thus broadening the appeal of the tool.[9] To this end, our present monitoring boundary implementation over TCP should facilitate technology-agnostic monitoring of multi-technology systems.

The elevation of the monitoring boundary at the specification level lends itself to further conceptual development. We plan to extend our technique to handle *dynamic* monitor partitioning across the system/remote-sides that can reconfigure itself as the system evolves, thus adapting to changes such as fluctuating system loads. The monitoring boundary also gives scope for various static analyses that can be carried out on our existing polyLarva scripts so as to obtain *automated* partitioning and placement of monitors across the boundary.

References

1. Arnold, M., Vechev, M., Yahav, E.: Qvm: an efficient runtime for detecting defects in deployed systems. SIGPLAN Not. 43, 143–162 (2008)
2. Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.): First International Conference, RV 2010, Lecture Notes in Computer Science, vol. 6418. Springer (2010)
3. Bodden, E., Chen, F., Rosu, G.: Dependent advice: a general approach to optimizing history-based aspects. In: Proceedings of the 8th ACM international conference on Aspect-oriented software development. pp. 3–14. AOSD '09, ACM (2009)
4. Bodden, E., Hendren, L., Lhoták, O.: A staged static program analysis to improve the performance of runtime monitoring. In: ECOOP 2007 - Object-Oriented Programming, 21st European Conference. Lecture Notes in Computer Science, vol. 4609, pp. 525–549. Springer (2007)
5. Colombo, C., Pace, G.J., Schneider, G.: Larva — safer monitoring of real-time java programs (tool paper). In: Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM). pp. 33–37. IEEE (2009)
6. Dwyer, M.B., Diep, M., Elbaum, S.: Reducing the cost of path property monitoring through sampling. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. pp. 228–237. ASE '08, IEEE (2008)
7. Geilen, M.: On the construction of monitors for temporal logic properties. Electr. Notes Theor. Comput. Sci. 55(2), 181–199 (2001)
8. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: A run-time assurance approach for java programs. Formal Methods in System Design 24, 129–155 (2004)
9. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. International Journal on Software Techniques for Technology Transfer (2011), to appear
10. Sen, K., Rosu, G., Agha, G.: Generating optimal linear temporal logic monitors by coinduction. In: Asian Computing Science Conference (ASIAN'03). Lecture Notes in Computer Science, vol. 2896, pp. 260–275. Springer (2004)

[9] At the time of writing, polyLarva supports monitoring of systems written in Java.