Challenges in Fault-tolerant Distributed Runtime Verification

Borzoo Bonakdarpour[1], Pierre Fraigniaud[2], Sergio Rajsbaum[3], and Corentin Travers[4]

[1] McMaster University, Canada, borzoo@mcmaster.ca
[2] LIAFA, Paris, Pierre.Fraigniaud@liafa.univ-paris-diderot.fr
[3] Instituto de Matemáticas, UNAM, Mexico rajsbaum@im.unam.mx
[4] U. Bordeaux, France, travers@labri.fr

Abstract. *Runtime Verification* is a lightweight method for monitoring the formal specification of a system (usually in some form of temporal logics) at execution time. In a setting, where a set of distributed monitors have only a partial view of a large system and may be subject to different types of faults, the literature of runtime verification falls short in answering many fundamental questions. Examples include techniques to reason about the soundness and consistency of the collective set of verdicts computed by the set of distributed monitors. In this paper, we discuss open research problems on fault-tolerant distributed monitoring that stem from different design choices and implementation platforms.

1 Introduction

Runtime verification (RV) is concerned with monitoring software and hardware system executions. It is used after deployment of the system for ensuring reliability, safety, and security, and for providing fault containment and recovery. Its essential objective is to determine at run time, whether the system is in a legal or illegal state, with respect to some specification.

1.1 The RV framework

An RV framework is essentially a two-layered system: the *underlying system*, and the *monitoring system*, interacting through two components. First, a *correctness specification* ϕ, which defines, at any moment during run time, if the underlying system is behaving correctly. Second, a *communication interface*, which is the subsystem stating how both layers communicate with each other. The communication is mainly one way, the monitoring system continuously gets information about the state of the underlying system. Although eventually there should be some way of getting feedback into the underlying system, for the whole setting to be useful. As soon as a violation of the legality of the execution is revealed recovery code can be executed for bringing the system back to a legal state, for runtime enforcement. For example, the recovery code can reboot the system, or release its resources. Runtime enforcement aims at guaranteeing desired

behaviors, e.g. [8]. The main communication is nevertheless upwards, and it defines what can be observed about the underlying system, by which means, how frequently and how reliably.

One source of difficulties in RV is the *decoupled design* of its two layers. The whole RV system is designed and built by two different parties, one of which does not know about the other. The underlying system is designed, and deployed, perhaps without even considering the possibility that a monitoring system will be built later on. Thus, with no concerns about the communication interface; no provisions for exporting data to a monitoring layer and neither for receiving feedback from it. Also, the correctness specification is not designed with a monitoring system in mind. It may be stated for infinite traces, while at run time, only finite traces are available. There may not exist at all a specification of whether a finite trace is correct or not. In addition, classic specification languages such as LTL are designed for infinite traces. Thus, a main concern in RV is to design finite trace semantics for RV, often based on LTL, and to study which properties are monitorable at run time.

In the simplest scenario, both the underlying system and the monitoring system are considered to be centralized, sequential processes. The underlying system produces *runtime traces*, which are finite sequences of *samples*, where each sample contains relevant information for the monitor about the current underlying system state. The monitoring system receives the sequence of samples as input. Perhaps each sample is triggered by an event in the underlying system or it is requested by demand of the monitoring system. In any case, the monitoring system uses the sequence of samples to successively expand a runtime trace. The goal of the monitoring system is, for each such trace, to emit a *verdict* about the valuation of ϕ. It maybe that a clear violation of correctness is observed on a trace α, or that no violation is seen and cannot happen in the future. But in general, without knowing the future, the problem of what verdict to emit arises. The three-valued logic LTL_3 [4] suggests to use the value '?' as a verdict in such a situation, while RV-LTL [3] refines this inconclusive verdict in two, possibly true $\top_p$ and possibly false $\bot_p$ verdicts, and more generally, LTL_K is a family of $(2k+4)$-valued logics, for $k \geq 0$ [7]. For each $k \geq 0$, the kth instance of the family has $2k+4$ truth values, that intuitively represent a *degree of certainty* that the formula is satisfied [7].

1.2 Decentralized RV

In this paper, we are concerned with RV when the underlying system is distributed. It could consist of computer hardware or other interacting machines, or a set of collaborating software components or a mix of both, for example, an aircraft. In this case, it makes sense to deploy monitors at different locations of the underlaying system. Passing messages to a *central* monitor at every event leads to communication bottlenecks, a single point of failure, and delays from far away components of the underlying system to the central monitor. Therefore, recent contributions e.g., [5, 6, 17, 18] on RV of distributed systems assume a set of n monitors observing the behavior of the underlying system, with benefits

such as replication (e.g. tolerate failures of the monitors themselves, or failures of sensors) or locality (e.g. a monitor observing some region or component of the underlying system). The monitors communicate with each other, to be able to tolerate failures of the monitors themselves. Also, to be able to evaluate a correctness condition that may depend on samples by several monitors. In short, in *decentralised RV*, both the underlying and the monitoring system are distributed systems.

The specific distributed RV setting depends first of all on which type of distributed system each one of its two layers is. A distributed system is defined by the asynchrony of processes, how they communicate with each other, and by the types of failures that may happen. One may divide them in two classes. Those where the processes can solve consensus, and those where they cannot. When the monitoring system is reliable enough to be able to solve consensus, monitors can exchange samples, and compute a snapshot representing the underlying system state. Then, each one locally can evaluate ϕ on this global state, and emit a verdict. Many papers exists on this scenario, where the concerns are about efficiency, perhaps distribution of the correctness formula, snapshot computation, reaching consensus on the snapshot as fast as possible, and so on, to build a monitoring layer that is as lightweight and quick as possible.

1.3 Distributed RV

In this paper we focus on the issues that arise when the monitoring system itself is distributed, unreliable, and unable to solve consensus. We call this setting *distributed RV*, to distinguish it from the more general decentralised RV. Instead of efficiency considerations, we discuss modeling and computability difficulties that arise in distributed RV. Many other issues related to efficiency in reliable decentralised RV, although they are challenging and important, they are somewhat better understood.

The purpose of this note is to discuss some of the new, fascinating issues that arise in distributed RV, and to discuss some of the existing work, which is far less than the work done on centralized RV. For some of the issues we discuss some possible solutions, for others we leave them open for discussion.

1. How to model an unreliable distributed RV system?
2. How is the correctness of the underlying system specified?
3. How many different verdicts are needed, and what is their meaning?
4. What is the meaning of a set of verdicts emitted by the monitors?
5. What is the process of giving feedback to the underlying system?

2 Challenges in Distributed Monitoring

We present a concrete distributed RV setting, very simple, but that serves as a basis to discuss the issues mentioned above. There are many other possible settings, but even in this very ideal setting already interesting difficulties appear.

2.1 A Distributed RV System

Challenge: how to make sure the monitors take samples about the same global state of the underlying system?

We now describe a distributed monitor system with two properties: monitors are unable to solve consensus, and it is simple. This system is very weak, but computability issues here extrapolate to other, stronger models, such as message-passing where at most t monitors may fail by crashing, Byzantine failures and others [14]. In particular, the following shared-memory model can be simulated in a message passing system if less than half of the monitors can fail[5] [1].

Assume that the system under inspection produces a finite trace $\alpha = s_0 s_1 \cdots s_k$ of global states. The trace is inspected with respect to some correctness specification expressed by a formula φ and a set of monitors $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$. The monitors communicate with each other by writing and reading shared memory registers. They execute the following algorithm.

For every $j \in [0, k-1]$, between each s_j and s_{j+1}, each monitor M_i:

1. takes a sample which results in a *partial* observation of s_j, denoted $\mathcal{S}_i(s_j)$;
2. repeatedly communicates with other monitors through a shared memory, and
3. emits a verdict about the correctness of $\alpha = s_0 s_1 \cdots s_j$.

If the samples $\mathcal{S}_i(s_j)$ *cover* s_j, i.e. collectively have all the information about s_j, then if the monitors could gather all their samples, they could locally evaluate φ. Instead, we assume each monitor $M_i \in \mathcal{M}$ is a sequential asynchronous process. It runs at its own speed, that may vary along with time. In step 2, monitors communicate with each other as much as they want, but a fixed, finite number of times. Namely, the code that each monitor executes in step 2 consists of N write and read instructions (no waiting for events in other monitors). We assume the monitors can communicate fast enough so that the underlying system changes to its new state s_{j+1} only once each monitor has executed its N instructions.

The monitor system corresponds to a *layered asynchronous wait-free read-/write shared memory* model [16]. In such a model, monitors cannot agree on a common view about s_j, due to the impossibility of solving consensus [13]. Additional details about fault-tolerant distributed computing appear in standard textbooks e.g. [2].

2.2 Distributed Correctness Specifications

Challenge: how to define a correctness specification in a way that can be evaluated on partial views of the global underlying system state?

[5] We work in shared memory because then we can consider runs composed of *any* interleaving of monitor operations, facilitating analysis. Also to including the extreme case where any number of monitors may fail. In message passing partitions may happen if half of the monitors can fail.

An execution of the monitoring system starting in state s_j (step 2 above), consists of an interleaving of the N operations of each one of the n monitors, and furthermore, any interleaving is possible (thus, it is as if monitors may crash, but technically it is not necessary to assume crashes). Consider a monitor M_i, and such an execution α. There is a subset of monitors from which M_i learns their samples: all of the monitors that write their samples to shared registers before M_i reads those registers. Therefore, M_i has to be able to emit its verdict based only on a partial view about s_j. Given that the code executed by the monitors is wait-free, in an extreme case, it is possible that M_i finishes its code before all the other monitors, and its partial view could consists of only its own sample.

It follows from the above discussion there has to be a way for a monitor to emit a verdict based on a partial view of the state. One approach is to assume a list of all possible partial views is given, including a predicate stating if the partial view is correct or not. This type of specification is called a *distributed language* in [10–12], and is used without any formal semantics.

Another approach is to assume that the specification φ was designed for (full) global states of the underlying system only, and use an *extension function* which somehow completes the partial view to a full state that *could* have occurred in the underlying system, as in [7, 9]. Care must be taken to ensure that different monitors use extensions on their own partial views that are somehow compatible.

Also, the difficulty remains even in the case that the monitors are much more synchronous and reliable. The partial views will be much more complete, but still consensus is impossible if the monitors may miss at least one sample. Notice that even if monitors are fully synchronous, there are lower bounds on how many rounds of communication are needed to solve consensus [2, 16], and thus there may not be enough time to reach consensus in between s_j and s_{j+1}.

2.3 Different Verdicts

Challenge: given that different perspectives about the underlying system state are unavoidable, how should they be used?

Given that in a wait-free distributed monitoring system it is impossible for the monitors to solve consensus, it is unavoidable that different verdicts are emitted. It was shown in [12], that as φ gets more and more "complex," more and more different verdicts have to be used.

Let us consider the following motivating example e.g. [3], of a system in which *requests* are sent by clients, and *acknowledged* by servers. The system is in a legal state if and only if (1) all requests have been acknowledged, and (2) every received acknowledgment corresponds to a previously sent request. Each monitor i is aware of a subset R_i of requests that has been received by the servers, and a subset A_i of acknowledgments that has been sent by the servers. To verify legality of the system, each monitor M_i communicates with other monitors in order to produce some verdict o_i. In a traditional setting of decentralized monitoring, it may be required that the monitors produce opinions $o_i \in \{\text{true}, \text{false}\}$ such that, whenever the system is not in a legal state, at least one monitor produces the

opinion false. It was shown in [12] that even if there is only one possible request and one possible acknowledgment, already three different verdicts are needed (with wait-free monitors).

To prove the lower bound on the number of verdicts required, a minimal *consistency* requirement is assumed in [12], stating that the set of verdicts should distinguish correct from incorrect traces. Namely, if a set of verdicts S is emitted on a correct trace, the same set should never be emitted on an incorrect trace.

2.4 Semantics of Verdicts

Challenge: which logic, which semantics should be assigned to an opinion, and which formulas are monitorable under a given distributed model?

Even in centralized RV, it has been observed that we need more than binary verdicts (i.e., true/false) to evaluate a finite trace. In RV-LTL [3] four truth values $\mathbb{B}_4 = \{\top, \bot, \top_p, \bot_p\}$ are used. These values identify cases where a finite execution (1) permanently satisfies, (2) permanently violates, (3) presumably satisfies, or (4) presumably violates an LTL formula. A multi-valued family of temporal logics refining RV-LTL was proposed in [7], each one with $2k+4$ values, denoted LTL_K, for $k \geq 0$. In particular, LTL_K coincides with RV-LTL when $k = 0$. The syntax of LTL_K is identical to that of LTL. Its semantics is based on FLTL [15] and LTL_3 [4], two LTL-based finite trace semantics for RV. For each $k \geq 0$, the kth instance of the family has $2k+4$ truth values, that intuitively represent a *degree of certainty* that the formula is satisfied. In particular, when $t = 2$, the set of truth values $\mathbb{B}_6 = \{\top, \bot, \top_0, \bot_0, \top_1, \bot_1\}$, can be used to monitor a request/acknowledgment formula with two types of requests and acknowledgments. It evaluates to: $\bot_0$ (presumably false with low degree of certainty) in a finite execution that only contains r_1, to $\top_0$ (presumably true with the same degree of certainty) in an execution that includes r_1 and a_1, to $\bot_1$ (presumably false with higher degree of certainty) in an execution that contains r_1, a_1, and r_2, and to $\top_1$ (presumably true with higher degree of certainty) in an execution that contains r_1, a_1, r_2, and a_2.

The LTL_K logic gives a formal semantics for the verdict of each monitor. It remains an open question how to get a formal semantics for *collections* of opinions emitted by the monitors of a decentralized system.

3 Conclusion

In this short note we are unable to discuss many other interesting issues related to distributed RV. The mechanisms of giving feedback from the monitors to the underlying system have not been studied. Given that each monitor emits a verdict, how and when are the collective verdicts used to give feedback to the underlying system? The simplest approach it to send all the verdicts to a human or machine control center, that decides what to do, but other, more distributed approaches should be studied. The work on wait-free distributed RV [10–12]

has focused on studying only one iteration of the algorithm discussed above, namely, where monitors take only one sample, and emit a verdict only once. Many technical issues arise in the general case of repeatedly taking samples.

There are already quite a few papers studying situations where more than one monitor is used, from different angles and using different techniques, and the *Bertinoro Seminar on Distributed Runtime Verification*, May 2016, was devoted to discuss these ideas.

4 Acknowledgment

This work was partially sponsored by Canada NSERC Discovery Grant 418396-2012 and NSERC Strategic Grants 430575-2012 and 463324-2014, Mexico UNAM-PAPIIT IN107714 and CONACYT-ECOS-NORD grants, as well as the French State, managed by the French National Research Agency (ANR) in the frame of the "Investments for the future" Programme IdEx Bordeaux - CPU (ANR-10-IDEX-03-02).

References

1. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, Jan. 1995.
2. H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley, 2004.
3. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.
4. A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14, 2011.
5. A. K. Bauer and Y. Falcone. Decentralised LTL monitoring. In *Proceedings of the 18th International Symposium Formal Methods (FM)*, volume 7436 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2012.
6. S. Berkovich, B. Bonakdarpour, and S. Fischmeister. Runtime verification with minimal intrusion through parallelism. *Formal Methods in System Design (FMSD)*, 46(3):317–348, 2015.
7. B. Bonakdarpour, P. Fraigniaud, S. Rajsbaum, D. A. Rosenblueth, and C. Travers. Decentralized asynchronous crash-resilient runtime verification. In *Proceedings of the 27th International Conference on Concurrency Theory (CONCUR)*, 2016. To appear.
8. Y. Falcone. *You Should Better Enforce Than Verify*, pages 89–105. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
9. Y. Falcone, T. Cornebize, and J.-C. Fernandez. *Efficient and Generalized Decentralized Monitoring of Regular Languages*, pages 66–83. Springer, 2014.
10. P. Fraigniaud, S. Rajsbaum, M. Roy, and C. Travers. The opinion number of set-agreement. In *18th Int. Conference on Principles of Distributed Systems (OPODIS)*, LNCS 8878, pages 155–170. Springer, 2014.
11. P. Fraigniaud, S. Rajsbaum, and C. Travers. Locality and checkability in wait-free computing. *Distributed Computing*, 26(4):223–242, 2013.

12. P. Fraigniaud, S. Rajsbaum, and C. Travers. On the number of opinions needed for fault-tolerant run-time monitoring in distributed systems. In *5th Int. Conference on Runtime Verification (RV)*, LNCS 8734, pages 92–107. Springer, 2014.
13. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Jan. 1991.
14. M. Herlihy, D. Kozlov, and S. Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann-Elsevier, 2013.
15. Z. Manna and A. Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.
16. Y. Moses and S. Rajsbaum. A layered analysis of consensus. *SIAM J. Comput.*, 31(4):989–1021, Apr. 2002.
17. M. Mostafa and B. Bonakdarpour. Decentralized runtime verification of LTL specifications in distributed systems. In *Int. Parallel Dist. Proc. Symp. (IPDPS)*, 2015.
18. K. Sen, A. Vardhan, G. Agha, and G. Rosu. Decentralized runtime analysis of multithreaded applications. In *Int. Parallel Dist. Proc. Symp. (IPDPS)*, 2006.