On Certifying Distributed Algorithms: Problem of Local Correctness

Kim Völlinger

On Certifying Distributed Algorithms: Problem of Local Correctness

K. Völlinger

Humboldt University of Berlin, Germany
voellinger@hu-berlin.de

Abstract. A *certifying* distributed algorithm (CDA) is a runtime verification method for distributed systems. Additionally to each output, a CDA computes a *witness* – a correctness argument for the particular output. If the witness is verified at *runtime*, the output is correct. The output is *distributed* over the system with each component holding its part of the distributed output.
In this paper, we investigate the case where the verification at runtime fails. Assume one component computes its part of the distributed output incorrectly. As a consequence, the distributed output is incorrect and the verification fails. Some components may still hold a correct part of the output. That is why we introduce the problem of *local correctness* of a component: *Is a component's part of the output correct?* As a case study, we investigate local correctness for a CDA computing shortest paths as used in distance-vector routing.

1 Introduction

A major problem in software engineering is assuring the *correctness* of a distributed system. A distributed algorithm runs on a distributed system where components communicate with each other in order to solve a common problem. The correctness of a distributed algorithm and of its implementation is crucial for the correctness of a distributed system. While formal verification is often too costly, testing is not sufficient if the system is of critical importance. Runtime verification tries to bridge this gap; it is not complete since verification at runtime can fail but it is a formal method based on mathematical reasoning.

Certifying Distributed Algorithms. A *certifying distributed algorithm* (CDA) is a runtime verification method. In order to verify its input-output pair (i, o), a CDA additionally computes a *witness* w such that if a *witness predicate* holds for the triple (i, o, w), the input-output pair (i, o) is correct [10]. A distributed *checker* algorithm decides the witness predicate at *runtime* [9]. To enable distributed checking, the witness predicate is *distributable*, i.e. a property in the system is expressed by stating properties for each component [7]. As an example of a witness, consider a distributed algorithm where the components of a network decide whether the network graph itself is bipartite. In case of a non-bipartite network graph, an odd cycle in the graph is a witness since an odd cycle is not bipartite itself. The witness predicate states that an odd cycle exists in

the network. A distributable variant of this witness is given in [7]. With a CDA, a user does not have to trust the distributed algorithm, its implementation or execution but only the checker. With a well-chosen witness, the checker is simple and its verification feasible [9,8]. The idea of a CDA is to adapt the underlying algorithm of a program at design-time such that it verifies its input-output pair at runtime. In the typical setup of runtime verification, a system is instrumented to send outputs to a trusted monitor which decides if a given property holds [3]. Analogously, a CDA is instrumented to compute a witness and send it to the checker which decides if the input-output pair is correct.

Contribution of this Paper. The input, output and witness of a CDA are distributed over the system with each system's component holding its part. In this paper, we assume that one component computes its part of the output incorrectly. As a consequence, the distributed output is incorrect and verification fails at runtime, i.e. the checker rejects. However, the outputs of some components computed may still be correct. That is why we introduce the problem of *local correctness* of a component: *Is a component's part of the output correct?* As a case study, we investigate local correctness for a CDA computing shortest paths [10] as used in distance-vector routing [6].

Related Work. Certifying *sequential* algorithms are well established [5] but there is little work on certifying *distributed* algorithms [10,8,7,9,1]. However, CDAs can be classified as a distributed and choreographed runtime verification approach since the checker is a distributed algorithm, as well as a synchronous runtime verification approach since the system waits for the checker to accept [2]. To our knowledge, there is no work on the problem of local correctness.

2 Preliminaries: Certifying Distributed Algorithms

As distributed systems, we consider *networks* that are *asynchronous* (i.e. no global clock), *static* (i.e. unchanged topology) and *id-based* (i.e unique identifiers). We model the communication topology of a network as a connected undirected graph $N = (V, E)$: a vertex represents a component, an edge a channel. A *distributed algorithm* describes for each component a reactive algorithm such that all components together solve one problem (e.g. leader election or coloring) [4,6]. The input i of a distributed algorithm is distributed such that each component $v \in V$ has its input i_v and $i = \cup_{v \in V} i_v$; analogously for the output. A CDA computes a distributed *witness* w additionally to its input-output pair (i, o) such that if a predicate (the *witness predicate*) holds for the triple (i, o, w), the pair (i, o) is correct [10]. We call a predicate that is defined over a component's input, output and witness a *local predicate*. A predicate Γ is *universally distributable* with a local predicate γ if for all triples (i, o, w) holds $(\forall v \in V : \gamma(i_v, o_v, w_v)) \longrightarrow \Gamma(i, o, w)$, and *existentially distributable* if $(\exists v \in V : \gamma(i_v, o_v, w_v)) \longrightarrow \Gamma(i, o, w)$. A predicate is *distributable* if one of the former applies, or if the predicate is implied by conjuncted and/or disjuncted universally/existentially distributable predicates [7]. The witness predicate is distributable, and can be decided by a distributed *checker* algorithm at runtime.

Each component v has a checker algorithm that decides all local predicates over (i_v, o_v, w_v). Using a spanning tree, the checkers of the components aggregate the evaluated local predicates upwards and combine them by logical conjunction or disjunction depending on whether the according predicate is universally or existentially distributable; the root decides the witness predicate by combining the evaluated distributable predicates [9]. Hence, if the distributed checker accepts, the distributed input-output pair (i, o) is correct. The user of a CDA does not have to trust the actual algorithm but the distributed checker. The simplicity of the checker depends on the choice of the witness. Using the framework proposed in [8,9], an implemented distributed checker can be verified.

Certifying Distributed Shortest Path Computation. A certifying variant of the distributed Bellman-Ford Algorithm computing shortest paths in a network is described in [10]. We assume a network $N = (V, E)$ where the channels have positive costs $cost : E \to \mathbb{R}_{>0}$. The length of a path is the sum of the costs of its edges. We assume one special vertex, the source s. The length of a shortest path from the source to a vertex v is the *distance* of v. A function $D_s : V \to \mathbb{R}_{\geq 0}$ is a distance function for s iff [5]:

$$D_s(s) = 0 \tag{1}$$

$$\text{for each } (u, v) \in E : D_s(v) \leq D_s(u) + cost(u, v) \tag{2}$$

$$\text{for each } v \in V, v \neq s \text{ there exists } (u, v) \in E : D_s(v) = D_s(u) + cost(u, v) \tag{3}$$

The distributed Bellman-Ford algorithm [6] solves the shortest path problem; each component v computes its distance v_{Ds} to the source s. Note that we distinguish the computed distance v_{Ds} from the actual distance $D_s(v)$ since the computed distance could be incorrect. In the certifying variant, each component v additionally computes its part of the witness: the computed distances of its neighbors, and a neighbor with which the property (3) holds – its parent in the shortest path tree. The witness predicate is satisfied iff the properties (1)-(3) hold. The checker of each component v decides the properties (1)-(3) as local predicates for v. The witness predicate is universally distributable. Hence, if the checker of each component accepts, the distributed checker accepts and all computed distances are correct.

3 Problem: Local Correctness of a Component

We assume a CDA with a distributed checker for some problem. We know if the distributed checker accepts, the particular distributed input-output pair is correct. We assume that one component is faulty and computes its part of the output incorrectly. As a consequence, the distributed output is incorrect. Thereby, the witness predicate does not hold and the distributed checker rejects. Hence, the verification at runtime fails. A solution could be to repeat the whole computation or to use another distributed algorithm. However, while the output of some components may be affected by the incorrect output of the faulty component, other components may still hold a correct part of the output. That is why we introduce

the problem of *local correctness* of a component: *Is the output of a component correct?* In some scenarios, it might be interesting to identify and keep correct parts of the output rather than repeating the computation. Hence, we think local correctness is worth to investigate. Note that local correctness requires that we know what correctness of a part of the distributed output means.

3.1 Case Study: Local Correctness in Shortest Paths Computation

We conduct a case study on local correctness for certifying shortest path computation as introduced in Section 2. We elaborate whether witness and checker are helpful in deciding local correctness. We assume that a component $v \neq s$ computes its output v_{Ds} incorrectly but with the right type (i.e. a positive number). Hence, $v_{Ds} \neq D_s(v)$, $v_{Ds} \geq 0$ and $v \neq s$. Each component $u \neq v$ computes its output u_{Ds} logically consistent to the faulty output v_{Ds}. Logically consistent means that for each component $u \neq v$ holds

$$u_{Ds} = 0 \; if \; u \; is \; the \; source \tag{4}$$

$$u_{Ds} \leq w_{Ds} + cost(u, w) \; for \; all \; neighbors \; w \; of \; u \tag{5}$$

$$u_{Ds} = w_{Ds} + cost(u, w) \; for \; at \; least \; one \; neighbor \; w \tag{6}$$

The properties (4)-(6) are equal to the properties (1)-(3) characterizing the distance function except that we are stating them for the *computed* outputs and only for the non-faulty components. The computed distances represent the distance function D_s iff the characterization properties hold for all components. Since v's output is incorrect, the output u_{Ds} can be correct ($u_{Ds} = D_s(u)$) or incorrect ($u_{Ds} \neq D_s(u)$) even though the properties (4)-(6) hold for u.

Identifying the Faulty Component using the Checker. Since the witness predicate is universally distributable, the distributed checker rejects iff at least one checker of a component rejects. The checker of the faulty component v rejects. If $v_{Ds} > D_s(v)$, the inequality (5) does not hold. There exists a neighbor w of v such that $D_s(v) = D_s(w) + cost(v, w)$. Hence, $v_{Ds} > D_s(w) + cost(v, w)$. In case of $v_{Ds} < D_s(v)$, the equality (6) does not hold.

For each component $u \neq v$, the checker of u accepts. Since u's output is logically consistent, the properties (4)-(6) hold for u. We can identify the faulty component by the rejection pattern of the distributed checker since exactly the checker of v rejects. Moreover, the checker of v can even decide whether v computed its distance too great or too small.

Fault Propagation. In the following, we investigate the local correctness of a component by studying fault propagation into several subnetworks. To exemplify, we consider the network N and a partitioning in the subnetworks I-VI highlighted in gray boxes as shown in Fig. 1. For simplicity, we omit the cost of a channel in the illustration. We argue under which circumstances the components of a subnetwork hold a correct part of the output. For our reasoning, we use arguments about the checker, witness and topology. We chose the topology of the network such that we can illustrate all topology-based arguments. However, arguments depending on the checker and witness are topology independent.

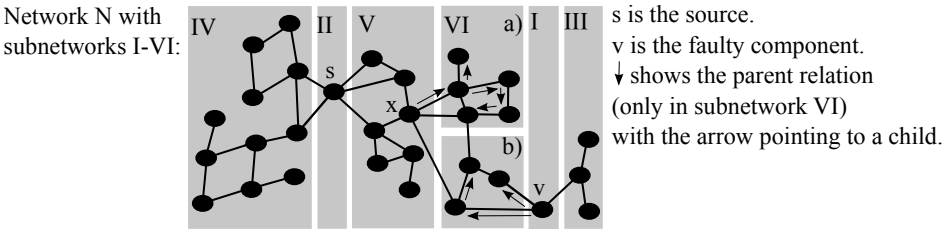

Fig. 1. Example network to illustrate fault propagation into subnetworks. Costs of channels are omitted. The parent relation is shown only in subnetwork VI.

Subnetwork I and II. Subnetwork I contains only the faulty component v, and by assumption, v's output is incorrect. Moreover, v is not the source s which is the only component in subnetwork II. The source's output is logically consistent, and therefore the equation (4) holds. As a consequence, the source's output $s_{Ds} = 0$ is correct.

Subnetwork III. For each component u of the subnetwork III, the output u_{Ds} is incorrect since each path from s to u has to contain v. More precisely, a component u computes its distance to v correctly as $D_v(u)$. Hence, $u_{Ds} = D_v(u) + v_{Ds}$. If the output u_{Ds} would be correct, then $u_{Ds} = D_v(u) + D_s(v)$, and since each path from s to u contains v, it follows $v_{Ds} = D_s(v)$. A contradiction to our assumption $v_{Ds} \neq D_s(v)$. Thus, $u_{Ds} \neq D_s(u)$ for all components u of subnetwork III.

Subnetwork IV. For each component u of the subnetwork IV, the output u_{Ds} is correct. The output u_{Ds} could be only affected by the faulty output v_{Ds} if the shortest path from s to u would contain v. Such an s-u path would contain s at least twice. Since all costs of the channels and the faulty output v_{Ds} are positive, such a path would never be computed as a shortest path. Hence, the fault of v does not propagate into the subnetwork IV.

Subnetwork V. For each component u of the subnetwork V, its output u_{Ds} is correct if the component v computed its distance too great: $v_{Ds} > D_s(v)$. Note that $D_s(v) = D_x(v) + D_s(x)$, and hence, $v_{Ds} > D_x(v) + D_s(x)$. Moreover, for each component u holds $D_s(u) \leq D_s(x)$ due to the positive costs of channels.

Subnetwork VI. For the subnetwork VI, we additionally consider the computed shortest path tree indicated by the computed parent relation. Using the parent relation, we distinguish components which are non-descendants of component v from components which are descendants of v.

Subnetwork VI a. The subnetwork VI a) contains the non-descendants of v of the subnetwork VI. If the component v computed its output v_{Ds} too small ($v_{Ds} < D_s(v)$), then all components of subnetwork VI a) have the correct output. The reason is that these components did not choose the component v as ancestor even though v offered an even smaller distance than it actually has.

Subnetwork VI b. The subnetwork VI b) contains the descendants of v. Hence, a component u of the subnetwork VI b) has the output $u_{Ds} = D_v(u) +$

v_{Ds}. Hence, the output u_{Ds} is potentially faulty. In contrast to the outputs of the components of the subnetwork III, $u_{Ds} = D_s(u)$ could still hold since there could be an s-u path without v which is a shortest and has the same sum u_{Ds}. Furthermore, note that, in the case of $v_{Ds} > D_s(v)$, the parent relation in subnetwork VI b) is part of an actual shortest path tree since these components chose v as an ancestor even though v offered a greater distance than it actually has.

Cut Vertices. For the example network (Fig 1), we draw some conclusions based on the topology. The components s, v and x are cut vertices, i.e. their removal increases the number of connected components. Those cut vertices are particularly interesting for fault propagation in our case study. Arguments using cut vertex x work for any cut vertex that separates the source and faulty component into different connected components. There are algorithms to detect cut vertices in a network [11]. However, for a static network, cut vertices could be known by initialization.

Arbitrary Topology. Some networks have no cut vertices. The arguments based on the computed parent relation are independent of the topology. The parent relation is part of the distributed witness. Moreover, the distributed checker identifies the faulty component v, and indicates whether v computed its distance too great or too small – both independent of the topology. Hence, the witness and the checker help in deciding local correctness for some components.

4 Discussion

We introduced the problem of *local correctness* of a component. We investigated local correctness in the context of a CDA where the runtime verification of a distributed input-output pair fails due to a faulty component. In particular, we conducted a case study of a CDA computing shortest paths, as for example used in distance-vector routing. In order to tackle local correctness, we investigated how a fault propagates through a network. We decided local correctness in subnetworks using the distributed checker, the distributed witness and the topology. We consider investigating local correctness in the context of a CDA promising since the distributed witness gives additional insight, being an argument for the correctness of an input-output pair.

Future Work. The case study could be extended by allowing the source to be faulty or by having several faulty components. Another direction is to study other problems. To study local correctness for a problem, there has to be a specification about the correctness of a component's output. Such a specification does not always come as natural as for the shortest path computation. Assume the problem of *leader election* where the components elect a unique leader among them. It is not straightforward what a correct leader election of a single component is since agreement on a leader is part of the problem. By relaxing local correctness of a component's output to the correctness of the outputs of a subnetwork, local correctness would be probably interesting for more problems.

References

1. Akili, S., Völlinger, K.: Case Study on Certifying Distributed Algorithms: Reducing Intrusiveness. In: Lecture Notes in Computer Science: 8th IPM International Conference on Fundamentals of Software Engineering. Springer (2019), to appear
2. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime Verification for Decentralised and Distributed Systems. In: Lectures on Runtime Verification, pp. 176–210. Springer (2018)
3. Hallé, S.: When RV Meets CEP. In: Falcone, Y., Sánchez, C. (eds.) Runtime Verification: 16th International Conference, RV 2016, Madrid, Spain, September 23–30, 2016, Proceedings. pp. 68–91. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-46982-9_6
4. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)
5. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying Algorithms. Computer Science Review 5, 119–161 (2011)
6. Raynal, M.: Distributed Algorithms for Message-Passing Systems. Springer Berlin Heidelberg (2013)
7. Völlinger, K.: Verifying the Output of a Distributed Algorithm Using Certification, pp. 424–430. Springer International Publishing, Cham (2017), https://doi.org/10.1007/978-3-319-67531-2_29
8. Völlinger, K., Akili, S.: Verifying a Class of Certifying Distributed Programs. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10227, pp. 373–388 (2017), https://doi.org/10.1007/978-3-319-57288-8_27
9. Völlinger, K., Akili, S.: On a Verification Framework for Certifying Distributed Algorithms: Distributed Checking and Consistency. In: Baier, C., Caires, L. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 38th IFIP WG 6.1 International Conference, FORTE 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10854, pp. 161–180. Springer (2018), https://doi.org/10.1007/978-3-319-92612-4_9
10. Völlinger, K., Reisig, W.: Certification of Distributed Algorithms Solving Problems with Optimal Substructure. In: Calinescu, R., Rumpe, B. (eds.) Software Engineering and Formal Methods - 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9276, pp. 190–195. Springer (2015)
11. Xiong, S., Li, J.: An efficient algorithm for cut vertex detection in wireless sensor networks. In: Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems. pp. 368–377. ICDCS '10, IEEE Computer Society, Washington, DC, USA (2010), http://dx.doi.org/10.1109/ICDCS.2010.38