Evaluation of Distributed Query-Based Monitoring over Data Distribution Service

Márton Búr*, Dániel Varró∗†‡
*McGill University, Montreal, Canada
marton.bur@mail.mcgill.ca, daniel.varro@mcgill.ca
†MTA-BME Lendület Cyber-Physical Systems Research Group, Budapest, Hungary
‡ Budapest University of Technology and Systems Economics, Budapest, Hungary

Abstract—Safety assurance in critical IoT systems is increasingly important as those systems are becoming ubiquitous in various application areas, such as health care and transportation. Recently, novel runtime monitoring approaches started to adapt expressive rule- or query-based languages to capture safety properties, which are evaluated over runtime models. In this paper, we define two distributed query-based runtime monitoring strategies for IoT and cyber-physical systems built on top of a distributed runtime model deployed over the Data Distribution Service as standard communication middleware. We provide detailed scalability evaluation of our prototype implementation over a case study motivated by an open-source IoT demonstrator.

Index Terms—Runtime models, Distributed queries, Runtime Monitoring, Data Distribution Service (DDS)

I. INTRODUCTION

Motivation: Critical Internet-of-things and cyber-physical systems (CPS) [1]–[3] frequently need to provide autonomous behavior with complex interaction with its uncertain environment using intelligent data processing techniques over a heterogeneous computation platform. In such systems, runtime verification (RV) [4], [5] is frequently used to ensure safe operation by monitoring. Recent RV approaches like started to exploit rule-based [6], [7] or query-based [7] techniques over a richer (relational or graph-based) information model.

Runtime models (also known as models@run.time [8], [9]) provide such a rich knowledge representation to capture the runtime state of the application data, services and platforms in the form of typed and attributed graphs [10] to serve as a unifying semantic basis for various kinds of analysis techniques. For example, runtime models have been used for the assurance of self-adaptive systems (SAS) in [11], [12].

In a distributed CPS over a resource constrained platform, the underlying runtime model also needs to be distributed, and updated with high frequency driven by the incoming sensor information and the changes in network topology.

Problem statement: In our previous work [7], we demonstrated the feasibility of query-based runtime monitoring, i.e. to define runtime monitors captured in a highly expressive declarative graph query language executed over a distributed runtime model deployed to a physical CPS platform. However, this approach used an ad hoc runtime model management layer (using low-level sockets for communication) while only an initial performance evaluation was carried out with few participants and medium-sized models. In [13], we adapted distributed runtime models over a standard middleware of (real-time) data distributed service (DDS) [14].

Objectives: Our current work provides a scalability evaluation of different query evaluation strategies used for runtime monitoring over a distributed runtime model which can be deployed over resource-constrained fog computing [15] environment. Our experimental evaluation is carried out on a case study motivated by the MoDeS3 CPS demonstrator (Model-Based Demonstrator for Smart and Safe Cyber-Physical Systems) [16], which is an open source educational platform. In particular, this paper provides the following contributions:

- We define single executor (coordinator-driven) and multiple executor (decentralized) monitoring strategies for evaluating graph queries over distributed runtime models.
- We provide prototype implementation of both strategies over a standard industrial DDS middleware.
- We carry out detailed scalability evaluation of this prototype implementation for increasing model size, number of participants and certain configuration parameters.

II. PRELIMINARIES: DISTRIBUTED RUNTIME MODELS

This section provides the modeling foundations for query-based runtime monitoring. We revisit definitions related to distributed runtime models based on our previous work [7].

A. Domain-specific modeling languages

In domain-specific (modeling) languages (DSLs), a domain is often defined by a metamodel including main concepts such as classes, attributes, and relations captured in the form of graph models and a set of structural consistency constraints. A metamodel can be formalized as a vocabulary \( \Sigma = \{C_1, \ldots, C_n, A_1, \ldots, A_m, R_1, \ldots, R_k\} \) with a unary predicate symbol \( C_i \) for each class, a binary predicate symbol \( A_j \) for each attribute, and a binary predicate symbol \( R_k \) for each relation in the metamodel.

Example: Figure 1 shows a metamodel for the MoDeS3 demonstrator with Participants (identified in the network by hostID attribute) which host the DomainElements. A DomainElement is either a Train or RailroadElement. A Train has a speed attribute and the train is always located on...
a RailroadElement. Turnouts and Segments are RailroadElements with links to the left and right side RailroadElements. These left and right references are used to describe the actual connections between the different RailroadElements. A Turnout has references to RailroadElements that represent the straight and divergent directions.

B. Distributed runtime (graph) models

We carry out runtime monitoring over runtime models [7] which serve as rich knowledge representation of the observations, internal state and the environment of the system in operation. The runtime model is distributed along participants the physical system and connected via the network [13]. These participants process the data provided by their corresponding sensors, and they are able to perform edge- or cloud-based computations on the data. The runtime model management components are deployed and executed on the platform elements, thus resource constraints need to be respected during allocation. The communication middleware needs to ensure fast and reliable communication between the components.

Formally, a runtime model \( M = \langle Dom_M, I_M \rangle \) is a logic structure over \( \Sigma \) as in [17], where \( Dom_M \) is a finite set of objects and elementary data values (integers, strings, etc.) in the model. \( I_M \) is a 2-valued interpretation of predicate symbols in \( \Sigma \) defined as follows (where \( o_p, o_q \) represent objects and \( a_p \) represents an attribute value):

- **Class predicates:** If object \( o_p \) is an instance of class \( C_i \), then the interpretation of \( C_i \) in \( M \) evaluates to 1 denoted by \( [C_i(o_p)]^M = 1 \), and it evaluates to 0 otherwise.

- **Attribute predicate:** If there is an attribute of type \( A_j \) in \( o_p \) with value \( a_r \) in \( M \), then \( [A_j(o_p, a_r)]^M = 1 \), and 0 otherwise.

- **Reference predicates:** If there is a link of type \( R_k \) from \( o_p \) to \( o_q \) in \( M \), then \( [R_k(o_p, o_q)]^M = 1 \), otherwise 0.

In our distributed runtime model [7], each participant has up-to-date but incomplete knowledge about the distributed system. Moreover, we assume that each model object is exclusively managed by a single participant, referred to as the *host* of that element, which serves as the single source of truth. This way, each participant can make calculations (e.g. execute a monitor locally) based on its own view of the system, and it is able to modify the mutable properties of its hosted model elements. Formally, for a predicate \( P \) with parameters \( v_1, \ldots, v_n \), \( [P(v_1, \ldots, v_n)]^M \) denotes its value over the distributed runtime model \( M \) is stored by host \( p \).

**Example:** Figure 2 shows a snapshot of the distributed runtime model \( M \) for the MoDeS3 system. Participants deployed to three different physical computing units manage different parts of the system. The model captures the three participants (Participant 1 – Participant 3) deployed to the computing units, the domain elements (s1–s8, tu1, tu2, tr1, and tr2) as well as the links between them. Each participant hosts model elements contained within them in the figure, e.g. Participant 2 is responsible for storing attributes and outgoing references of objects s3, s4, s5, and tr1.

C. Real-time Data Distribution Service

Both model update and monitoring messages are sent between the components over a middleware using a publish-subscribe protocol that implements the real-time data distribution service (RDDS [18]). RDDS is an extension for the DDS standard [14] of the Object Management Group (OMG) to unify common practices concerning data-centric communication using a publish-subscribe architecture. Furthermore, the middleware also abstracts from the platform and network-specific details, and it can provide quality of service (QoS) and quality of data (QoD) guarantees. Thus, in the current paper, we assume reliable and timely delivery of messages by the underlying middleware.

III. QUERY-BASED MONITORING

A. Graph queries for specifying safety monitors

As proposed in [7], we rely on the VIATRA Query Language (VQL) [19] to capture the monitored properties. This declarative graph query language allows to capture safety properties on a high level of abstraction over the runtime model, which eases the definition and comprehension of safety monitors for engineers and avoids accidental complexity caused by additional platform-specific or deployment details. While graph queries can be extended to express temporal behavior [20], this work is restricted to safety properties where the violation of a property is expressible by graph queries.

The expressiveness of VQL converges to first-order logic with transitive closure, thus it provides a rich language for capturing a variety of complex structural conditions and dependencies. Any match (result) of a graph query over the runtime model highlights a violation of the safety property. Formally, a graph query \( \varphi(v_1, \ldots, v_n) \) can be evaluated over a runtime model \( M \) (denoted by \( [\varphi(v_1, \ldots, v_n)]^M \)) along a variable binding from variables to objects and data values in \( M \) (i.e., \( Z : \{v_1, \ldots, v_n\} \rightarrow Dom_M \)) in accordance with the semantic
rules defined in [17]. A variable binding \( Z \) is called a \textit{match} if query \( \varphi \) is evaluated to 1 over \( M \), i.e. \( \left[ \varphi(v_1, \ldots, v_n) \right]_M^Z = 1 \).

Example: In the railway domain, safety standards prescribe a minimum distance between trains on track [21], [22]. The \texttt{closeTrains} monitor definition captures a (simplified) description of the minimum headway distance to identify violating situations where trains have only limited space between each other. Technically, one needs to detect if there are two different trains on two different railroad elements, which are connected by a third railroad element. Any match of this pattern highlights track elements where passing trains need to be stopped immediately. Listing 1 shows the monitoring query \texttt{closeTrains} in textual VQL syntax and Listing 2 displays it as a graph formula.

```
pattern closeTrains {
    start : RailroadElement, end : RailroadElement | 
    Train.on(train_1,start); 
    Train.on(train_2,end); 
    train_1 != train_2; 
    RailroadElement.connectedTo(start, middle); 
    RailroadElement.connectedTo(middle, end); 
    start != end; }
```

Listing 1: Monitoring goal in the VIATRA Query Language

```
CloseTrains(start,end) = 
RailroadElement(start) \land RailroadElement(end) \land 
\exists train_1 : \forall train_2 \land \forall train_3 \land \forall train_4 \land 
Train.on(train_1,start), 
Train.on(train_2,end), 
Train.on(train_3,start), 
Train.on(train_4,end), 
\neg (train_1 = train_2), 
\neg (train_2 = train_3), 
\neg (train_3 = train_4), 
\neg (train_1 = train_4), 
\neg (train_2 = train_4), 
\neg (train_3 = train_4), 
\neg (train_1 = train_3), 
\neg (train_2 = train_3), 
\neg (train_1 = train_2), 
ConnectedTo(middle,end) \land 
\neg (start = end)
```

Listing 2: Monitoring goal as a formula

B. Monitor execution

Our system-level runtime monitoring framework is \textit{hierarchical} and \textit{distributed}. Monitors may observe the local runtime model of a participant, and they can collect information from runtime models of different participants. Moreover, one monitor may request information from other monitors, thus yielding a hierarchical network.

Monitors compute matches of a graph query \( \varphi(v_1, \ldots, v_n) \) along a \textit{search plan} by assigning model objects to variables \( v_1, \ldots, v_n \) and evaluating the predicate of the query. A search plan is an ordered list of search operations (e.g. checking type of objects, navigating along references) that traverses the runtime graph model in order to find all complete variable bindings satisfying the query condition. The search plan for \( \varphi(v_1, \ldots, v_n) \) also depends on the initial binding information for the input parameters (provided by the caller) as a variable with a fixed value can greatly reduce the search space.

Example: The pseudo code in Algorithm 1 sketches the implementation for finding all matches of \texttt{closeTrains} if input parameters \texttt{(start} and \texttt{end} in L1) are unbound. Unbound input parameters and variables that are not assigned a value are represented with \texttt{NULL}. In L2 the set \texttt{matches} serves as the container of the results and is initialized empty. In the following line (L3), the if-statement ensures that both input parameters are unbound. The search algorithm is shown in L4-L13. It starts with a loop that binds objects of type \texttt{RailRoadElement} to variable \texttt{start}. Lines 5-6 show navigation along the train reference starting from \texttt{start}. If an object of type \texttt{Train} is present on \texttt{start}, the search continues by navigating twice along the \texttt{connectedTo} reference in each possible way, again, from \texttt{start} and binding variable \texttt{end} (L7-L9). In L10-L11 the presence of a train is checked on \texttt{end}. Finally, if both \texttt{train_1} and \texttt{train_2} are bound to different train objects (L12), a match is found and in \texttt{(start, end)} is added to \texttt{matches} in L13.

```
Algorithm 1 Compute results of \texttt{closeTrains}

1: \textbf{input}: start, end
2: \textbf{matches} \leftarrow \emptyset
3: \textbf{if} start = NULL and end = NULL \textbf{then}
4: \quad for \texttt{start} in \{all RailroadElement objects\} do
5: \quad \qquad \texttt{train_1} \leftarrow \texttt{start}.train()
6: \quad \textbf{if} \texttt{train_1} \neq NULL \textbf{then}
7: \quad \quad \textbf{for} \texttt{middle} in \texttt{start}.connectedTo() do
8: \quad \quad \quad \textbf{for} \texttt{end} in \texttt{middle}.connectedTo() do
9: \quad \quad \quad \quad \textbf{if} \texttt{start} \neq \texttt{end} \textbf{then}
10: \quad \quad \quad \quad \quad \texttt{train_2} \leftarrow \texttt{end}.train()
11: \quad \quad \quad \quad \quad \textbf{if} \texttt{train_2} \neq NULL \textbf{then}
12: \quad \quad \quad \quad \quad \quad \textbf{if} \texttt{train_1} \neq \texttt{train_2} \textbf{then}
13: \quad \quad \quad \quad \quad \quad \quad \texttt{matches}.add((\texttt{start, end}))
```

The complete matching algorithm of \texttt{closeTrains} needs to handle initial variable bindings (e.g. \texttt{end} \neq NULL) which is omitted here for space considerations.

In distributed query-based monitoring, a significant challenge in executing search plans is to manage navigation along references and reading an attribute values of objects hosted by other participants. Additionally, we wish to allow that monitors initiated by an arbitrary participant can return all violations in the whole system. We have identified two possible approaches to support these requirements:

- \textbf{Single executor}: a single participant executes all steps of the search plan while accessing information stored at other participants, which can be achieved by a synchronous remote procedure call implementation over DDS. The request message conveys i) what reference/attribute value of ii) what object is requested and iii) by which participant, while the reply message encapsulates the i) object identifier and ii) an array of values.
- \textbf{Multiple executors}: the participant which initiates the execution of a monitor will not directly request information from other participants. Instead, if a remote object is encountered and its reference/attribute value is queried, the partial variable bindings are asynchronously passed to the other participant, which continues the execution of the search plan. The request message needs to contain i) an auto-generated unique request identifier, ii) the partial variable binding, iii) the next step in the search plan and iv) the requester participant ID. Once the other participant receives this message, it continues the matching and
finally, it returns all found matches to the requester in a reply message with i) the set of matches and ii) the request identifier.

IV. EVALUATION

We evaluated the scalability of query-based runtime monitoring using these execution strategies with respect to increasing (a) model size, and (b) number of participants, and (c) certain configuration parameters of the DDS middleware.

A. Benchmark setup

1) CPS monitoring benchmark: To assess the distributed runtime verification framework, we used the MoDeS3 railway CPS demonstrator [16] where multiple safety properties are monitored. They are all based on important aspects of the domain. The monitoring goals of interest are the following:

- **CloseTrains**: see example in subsection III-A.
- **Derailment**: a train approaches a turnout that is set to the opposite direction (causing the train to derail).
- **EndOfSiding**: a train approaches an end of the track.
- **TrainLocations**: highlight segments with a train.

Since the original runtime model of the CPS demonstrator has only a total of 49 objects, we scaled up the model by replicating the original elements (except for the computing units). This way we obtained models with 140 – 1.4M objects having similar structural properties as the original one.

2) Technical details: We synthesized the C++ monitors for the MoDeS3 case study. As DDS implementation, we built upon RTI Connext Professional, which is currently considered to be the leading Industrial IoT connectivity framework.

We used a quality of service profile called **High Throughput** defined in the RTI Connext DDS library which provides an initial configuration designed for applications that require reliable data streaming at a high data rate. This profile improves the performance by allowing the middleware to buffer application messages and send them together in a single transport layer message, which is a UDPv4 datagram in our case. We slightly tailored this profile to better fit monitoring purposes, namely i) we added a deadline to force flushing the content of the buffer to the network at least every 100ms to avoid "stuck" messages and ii) we disabled the shared memory transport to achieve more realistic network throughput with communication over UDPv4.

We ran our distributed monitors in isolation but within the same physical host. We used Docker\(^1\) to manage application containers, and assigned a dedicated virtual network interface to allow communication between the containers. The hosting computer was a single server machine with 32 CPU cores.

Every single data point in the diagrams of this section displays results of 20 consecutive runs. A data point is obtained by removing the lowest and highest measured value, then taking the average of the remaining 18.

---

\(^1\)https://www.docker.com/

---

B. Experiment results

We only provide an excerpt of our findings in this section, while a complete export of conducted measurement data is available online\(^2\).

1) Scalability wrt. model sizes: In our previous work, we conducted an initial evaluation of our query-based monitoring approach deployed to a network of six physical single board computers [7] with 43K objects. Figure 3 summarizes our results over a DDS middleware with 10 participants and multiple executors. It shows scalability up to 1.4 million objects with similar runtime as reported in [7], which highlights that graph pattern matching scales well when deployed over an industrial DDS middleware. However, the single executor strategy failed to scale by providing 3-4 orders of magnitude slower execution times (for only 2 participants and 14K objects).

2) Scalability wrt. number of participants: The next scenario investigates the impact of increasing the number of participants on query evaluation time. The results for **CloseTrains** and **Derailment** monitors are depicted in Figures 4a and 4b, respectively. The x-axis shows the increasing model sizes while the y-axis depicts execution time for 2, 5, 10 and 20 participants.

The results show negligible difference in execution times with increasing number of participants in case of the selected two monitors. This leads us to the conclusion that the system scales well for higher degrees of distribution.

3) Impact of batching DDS messages: The default setting in the High Throughput DDS profile allows application message buffering. This buffer is packed in a single UDPv4 datagram once it reaches 30kB and sent over the network. With this set of experiments, we investigated how this buffer size impacts query evaluation performance. The results are shown in Figure 5a for **CloseTrains** and in Figure 5b for **Derailment** with two and five participants.

Buffering messages slows down query execution for models with less than 14K objects. However, buffering can improve execution run time for certain monitors over models with more than 14K elements (e.g. in case of **Derailment**) and will not degrade run time in others (like in case of **CloseTrains**).

---

\(^2\)https://nimbur.github.io/cps-query/
V. RELATED WORK

We overview related papers covering distributed model queries, runtime verification of distributed systems and IoT monitoring frameworks.

Distributed model queries: The authors of [23] provide a toolset called GreyCat for maintaining and analyzing temporal graphs over arbitrary storage technologies, such as in-memory stores or NoSQL databases. The IncQuery-D framework [24] provides support for distributed incremental model queries. It builds an in-memory middleware layer on top of a distributed model storage system, and uses the Rete algorithm [25] for incremental maintenance of query results.

As a key difference, these distributed graph query evaluation techniques use a cloud-based execution environment, thus they are not directly applicable for a heterogeneous execution IoT platform with low-memory computation units.

Runtime verification of distributed systems: Classical runtime verification approaches supporting distributed systems [26]–[28] frequently use temporal logic for specifying safety properties and expected behavior, which excel in temporal properties over atomic predicates but they typically fail to express complex structural requirements. Evaluating such complex structural properties over a distributed data model is, in fact, a main added value of our query-based approach.

Brace [29] is a framework for distributed runtime verification of CPSs that imposes low runtime overhead on the system under monitor. However, global properties are monitored by a dedicated central entity, which is different from our solution, where any of the participants can initiate and collect monitoring results from the whole system.

IoT monitoring frameworks: A cloud-centric vision for Internet of Things is presented in [30], which is a dominant approach in healthcare IoT applications. Typically, data collection from the environment is carried out at the edge of the network, then intelligent processing is performed on a cloud platform [31]–[32]. However, our focus differs from this in two ways, namely i) all the processing is done on the network edge and ii) the monitoring focus is on the system properties itself, and not on the environment.

VI. CONCLUSION AND FUTURE WORK

In this paper, we defined two strategies for distributed query-based runtime monitoring over a distributed runtime model
used in the context of critical CPS and IoT applications. We used the standard Data Distribution Service as an underlying communication middleware for runtime monitors, which provides QoS and QoD guarantees. We carried out detailed scalability evaluations using our prototype (implemented in C++) in the context of an open educational IoT demonstrator, which showed favorable results both with increasing number of participants (up to 20 participants) and increasing model size (with 1.4 million objects).

As a future work, we are planning on enhance the single executor strategy by adding support to asynchronously request information about the runtime model of other participants. This would enable parallel search execution by utilizing the time that is now spent on waiting for remote calls to complete.

ACKNOWLEDGMENT

This paper is partially supported by MTA-BME Lendület Cyber-Physical Systems Research Group, the NSERC RGPIN-04573-16 project, and the McConnell Memorial Fellowship (as part of the MEDA program). The authors would like to thank Gábor Szárnyas the help on providing the evaluation environment.

REFERENCES