Journal of Internet Services and Applications

RESEARCH Open Access

Timed protocol analysis of interconnected mobile IoT devices

Georgios Bouloukakis[1*], Nikolaos Georgantas[2], Ajay Kattepur[3] and Valerie Issarny[2]

*Correspondence:
georgios.bouloukakis@telecom-sudparis.eu
[1]Télécom SudParis, Institut Polytechnique de Paris, France
Full list of author information is available at the end of the article

Abstract

With the emergence of the *Internet of Things* (IoT), application developers can rely on a variety of protocols and *Application Programming Interfaces* (APIs) to support data exchange between IoT devices. However, this may result in highly heterogeneous IoT interactions in terms of both functional and non-functional semantics. To map between heterogeneous functional semantics, middleware connectors can be utilized to interconnect IoT devices via bridging mechanisms. In this paper, we make use of the *Data eXchange* (DeX) connector model that enables interoperability among heterogeneous IoT devices. DeX interactions, including synchronous, asynchronous and streaming, rely on generic *post* and *get* primitives to represent IoT device behaviors with varying space/time coupling. Nevertheless, non-functional *time semantics* of IoT interactions such as data availability/validity, intermittent connectivity and application processing time, can severely affect response times and success rates of DeX interactions. We introduce timing parameters for time semantics to enhance the DeX API. The new DeX API enables the mapping of both functional and time semantics of DeX interactions. By precisely studying these timing parameters using timed automata models, we verify conditions for successful interactions with DeX connectors. Furthermore, we statistically analyze through simulations the effect of varying timing parameters to ensure higher probabilities of successful interactions. Simulation experiments are compared with experiments run on the *DeX Mediators* (DeXM) framework to evaluate the accuracy of the results. This work can provide application developers with precise design time information when setting these timing parameters in order to ensure accurate runtime behavior.

Keywords: Middleware, IoT interactions, Interoperability, Timed automata, UPPAAL, Statistical analysis

1 Introduction

The Service Oriented Architecture (SOA) paradigm allows heterogeneous components to interact via standard interfaces by employing standard protocols. These are principally based on the client/server interaction paradigm, where typically a client sends a request to a server and gets the response within a *timeout* period. The successful completion of such an interaction depends on: *i)* the server's reachability; and *ii)* the time needed to process the request, in comparison to the timeout period applied by the client.

On the other hand, the advent of paradigms such as the Internet of Things (IoT) [1] involves not only conventional services but also sensor-actuator networks and data feeds. IoT applications consist of devices (i.e., *Things*) that may employ a variety of middleware-layer protocols, such as MQTT, CoAP, ZeroMQ [2] and more, to interact with each other. These protocols may follow different interaction paradigms such as Client/Server (CS), Publish/Subscribe (PS), Data Streaming (DS), Tuple Space (TS), which determine the functional semantics of interactions. However, IoT applications are characterized by non-functional ***time semantics*** as well. In particular, exchanged data records may be valid or available for a limited *lifetime* (time-to-live) period. In addition, a considerable portion of IoT is mobile, which results to intermittently available data recipients. The latter, in conjunction with the data availability/validity, may affect the successful delivery of data in IoT applications.

The primary purpose of this paper is to model and analyze the aforestated time semantics in mobile IoT interactions. To deal with the Things' heterogeneity, we leverage the *Data eXchange* (DeX) connector model [3], which maps the end-to-end functional semantics of Things that employ heterogeneous middleware protocols. This is achieved via the DeX API that abstracts and unifies the four identified interaction paradigms (CS, PS, DS and TS) into four basic interaction types: *(i) DeX one-way*; *(ii) DeX two-way sync*; *(iii) DeX two-way async*; and *(iv) DeX streaming* (more details in Section 3). The DeX connector makes part of the DeXM framework, which enables interoperability among Things through the automated synthesis of *mediators*. In this paper, we extend the DeX API and connector model by introducing the following ***timing parameters***: *(i)* `lifetime` and `timeout` to qualify the availability/validity in time of data or requests ; *(ii)* `serve_time` to qualify the time it takes a server to process requests; and *(iii)* `time_on`/`time_off` to qualify the intermittent availability of data recipients. In our previous work [4], we modeled and analyzed the timing behavior of one-way interactions in service choreographies. In this paper, we generalize our modeling and analysis to one-way, two-way synchronous & asynchronous, and streaming interactions in IoT applications. This applies to systems that rely on not only any protocol following the CS, PS, DS and TS paradigms, but also any interconnection between them.

For our modeling and analysis, we use *Timed Automata* [5]. By relying on this formalism, we identify conditions for successful interactions and verify reachability and safety properties by employing the UPPAAL [6] model-checker. We further perform statistical analysis through the simulation of DeX interactions over multiple runs, and study the trade-off between interaction success rate and response time with varying timing parameters. To evaluate the accuracy of our simulation results, we compare them with experiments run with real protocol implementations on the *DeX Mediators* (DeXM) framework.

The approach introduced in this paper can be used by developers to compare between interaction types and paradigms, select among them, tune the timing parameters of the overlying application, and also do the previous when interconnection between heterogeneous protocols is involved. The main contributions of this paper are summarized as follows:

- We define *timed DeX interactions* that represent basic end-to-end IoT interaction types enhanced with explicit *time semantics*.

- We introduce formal timed models for DeX one-way, two-way synchronous & asynchronous, and streaming interactions.
- We verify formal conditions for successful DeX interactions.
- We study the interaction success rate vs. response time trade-off with varying timing parameters through the simulation of DeX interactions.
- We validate our findings by running experiments with real protocol implementations.

The rest of the paper is organized as follows. We provide a motivating use case scenario and a high-level overview of our approach in Section 2. Then, in Section 3, we present the various semantics of DeX interactions, and introduce our enhancement of the DeX API. We also briefly describe the DeXM framework that interconnects IoT devices via mediators. The model for timing analysis of DeX interactions is introduced in Section 4. This is formalized with timed automata and related verification of properties in Section 5. The results of our analysis through simulation experiments are presented in Section 6, which includes comparison with experiments with the DeXM framework. This is followed by conclusions in Section 8.

2 Overview

Motivating Use Case. The detection and management of traffic congestion in a city is a critical issue in order to avoid significant delays while driving a vehicle [7]. For this purpose, several intelligent systems have been developed. We can classify them into three categories leveraging: *(i) fixed-sensors* (vehicle detectors, traffic cameras, doppler radars, etc) that have been installed on existing infrastructure [8, 9]; *(ii) vehicle* (on-board) *devices* with GPS-based systems [10]; and, *(iii) smartphones* with embedded sensors (accelerometer, gyroscope) [11]. As depicted in Fig. 1, the combination of such intelligent systems can provide us an overall *Transport Information Management* (TIM) system in order to accurately estimate traffic conditions.

However, each of the above sensors/applications may rely on different protocol primitives, timing requirements and constraints when exchanging data as shown in Fig. 1. In particular, city-deployed, vehicle-device and smartphone sensed data are sent (periodically or not) to a broker system and then to the estimation service using the `post` primitive. To guarantee the freshness of provided information, notifications are maintained by the system for a (limited) `lifetime` period.

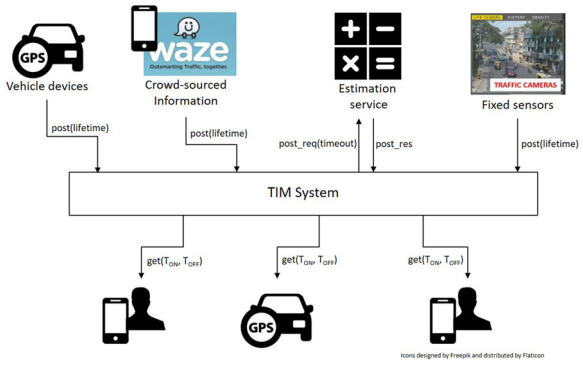

Fig. 1 Transport Information Management (TIM) system

Mobile smartphones and vehicle devices access the system using the `get` primitive to receive up-to-date transport information. They stay connected for an average period (T_{ON}) and then disconnect for resource saving purposes (T_{OFF}). Disconnections occur due to resource saving purposes (e.g., battery), based on the budget of the mobile user's data plan or based on the networking issues of mobile data recipients. Finally, the estimation service processes the collected data and provides back the estimated traffic to smartphones/vehicle devices using the `post_res` primitive after their posted request (`post_req`) that has to be completed within a `timeout` period.

Given the above scenario, this work aims to address the following research questions: *(i)* despite the possibly high disconnection periods of mobile users, can a system designer ensure the timely delivery of traffic data to all recipients? *(ii)* can a system designer configure the connected periods of mobile users in order to guarantee successful and timely delivery of messages? *(iii)* can the validity of data be leveraged and configured to ensure specific delivery success rates of data?

High-Level Approach. Based on the primitives and timing constraints of the TIM system, an application designer should be able to analyze and configure certain system aspects (user connectivity, message lifetime period, etc) in order to guarantee the appropriate system response time and delivery success rate.

In this paper, we provide a *time analysis approach* as shows in Fig. 2 that enables:

1. Analysing the application's functional and time semantics, i.e., primitives and timing parameters, based on the employed middleware protocol(s) and the timing behavior of the IoT application.
2. Ensuring successful interactions in the IoT application by relying on the formal conditions identified using *Timed Automata*.
3. Performing statistical analysis using the DeX timing parameters for enabling system tuning.

Accordingly, system designers are able to redefine timing parameters of the TIM system and derive the corresponding formal conditions for successful interactions. Finally, by relying on our statistical analysis methodology, we demonstrate in Section 6 how system designers are able to tune the TIM system in order to ensure specific time requirements.

3 Interconnecting heterogeneous things

To enable the development of applications in IoT spaces, middleware IoT protocols, such as MQTT, CoAP, XMPP, etc, can be leveraged. These protocols provide a number of features such as supporting different *Quality of Service* (QoS) guarantees and they can be

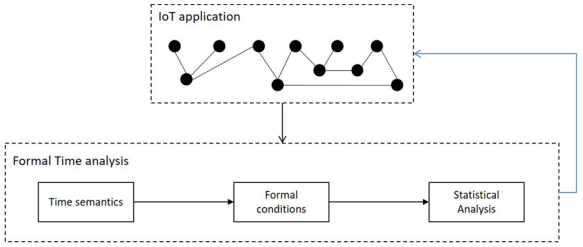

Fig. 2 Time Analysis Approach. Ensuring successful interactions into an IoT application

classified to an interaction paradigm (i.e., CS, PS, DS and TS). To express the different dimensions of coupling among communicating Things, we define the semantics of each interaction paradigm. We then leverage the DeX API [3], which is a set of primitives expressed as functions supported by the middleware. This API abstracts semantics of the basic interaction paradigms and therefore the semantics of the majority of existing IoT protocols. In this paper, we present a time-enhanced API that incorporates timing parameters of interactions such as timeout of requests, time validity of both messages (app-layer data) and control messages (e.g., subscriptions) and the Things intermittent connectivity. Finally, we explain how the DeX API is leveraged to model time-enhanced heterogeneous interactions in the IoT by relying on *mediator* software artifacts.

3.1 IoT semantics for data exchange

By relying on [12–14], semantics of interest include *space coupling*, *time coupling* and *synchronization coupling*. Space coupling determines how Things identify each other and, consequently, how interaction elements (such as messages) are routed from one Thing to the other. Time coupling essentially determines if Things need to be present and available at the same time for an interaction or if, alternatively, the interaction can take place in phases occurring at different times. Finally, synchronization coupling determines whether the initiator of an end-to-end interaction blocks or not until the interaction is complete; in the former case, the interaction is executed in a synchronous way between the interacting Things. To express synchronization semantics, but also other semantics of end-to-end interactions, we define four *interaction types* and six role types for the interacting Things:

1 one-way interaction: a Thing can take either the *sender* or the *receiver* role. The sender sends a piece of data without waiting for a response; the receiver will asynchronously get notified for the arrival of the element by setting a listening & callback mechanism.
2 two-way synchronous (sync) interaction: a Thing can take the *client* or *server* role. A synchronous interaction is blocking for the client and requires a prompt response from the server. Clients invoke a request on the server and then suspend their processing while they wait for a response for a specific `timeout` period.
3 two-way asynchronous (async) interaction: each Thing can take either the *client* or the *server* role. Clients initiate a request to a server and then continue their processing (non-blocking). The server handles the client's request using a callback and returns the response at some later point, at which time the client receives the response (also with a callback) and proceeds with its processing.
4 streaming interaction: a Thing can take either the *consumer* or the *producer* role. The consumer requests to establish a dedicated session with the producer. Once established, the producer sends multiple pieces of data that will asynchronously be received by the consumer. Depending on the middleware protocol, both Things or just the consumer can suspend, resume and terminate the session using the corresponding interaction elements.

As depicted in Fig. 3, middleware IoT protocols implement one or more of the above interaction types. For instance, in CS protocols, a client communicates directly with a server either by direct messaging (one-way) or by a remote procedure call (RPC, two-way) through an operation. In PS protocols, publishers publish events characterized by

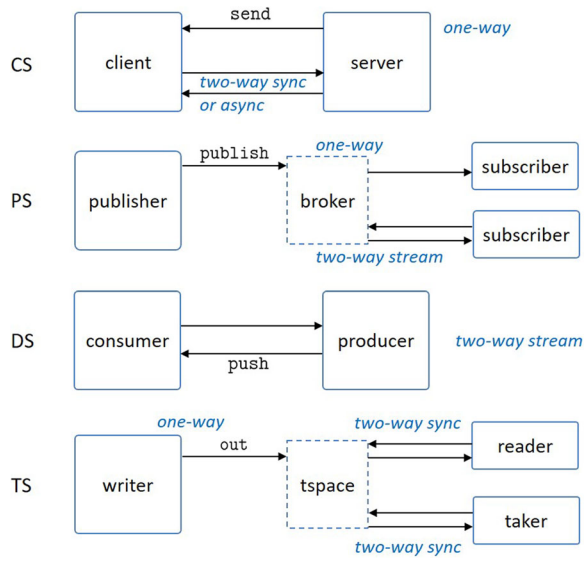

Fig. 3 CS, PS, DS and TS interaction paradigms and the supported interaction types

a specific *filter* (e.g., topic) to the broker. Subscribers may choose to check for pending events synchronously themselves (two-way synchronous) or set up a callback function that will be triggered asynchronously by the broker when an event arrives (streaming). In DS protocols, a consumer (typically) establishes a dedicated session with an *open stream* request sent to a producer. Upon the session's establishment, a continuous flow of data is pushed from the producer to the consumer. Finally, in TS protocols, multiple peers interact via an intermediate entity with a tuple space (*tspace*). Peers can write (*out*) data into the tspace and can also synchronously retrieve data from it, either by reading a copy or removing the data.

3.2 Data eXchange (DeX) API

The basic interaction types can be implemented using the corresponding library implementation of every IoT middleware protocol. In [3], we introduced the DeX connector that models the semantics of the majority of existing middleware protocols via the following high-level API primitives:

- (i) `post` employed by a Thing for sending data, also called *messages*, to one or more other Things.
- (ii) `get` employed by a Thing for receiving data.

In this paper, we enrich the DeX API [3] with the following timing parameters:

- `lifetime`: refers to emitted messages and characterizes both data availability/validity in time.
- `timeout`: allowed time period to complete a request-response synchronous interaction.
- `time_on`, `time_off`: instantaneous connected (`ON`) and disconnected (`OFF`) periods for receiving data.
- T_{ON}, T_{OFF}: average time periods of `time_on`, `time_off` applied from the app-layer.

In Table 1, we present the enhanced primitives per interaction type. Below we provide a brief description:

DeX one-way. In a DeX one-way interaction a *sender* entity sends messages using the `post` primitive with a validity period `lifetime`. At the receiver side, the `get` primitive is used to enable the reception of messages through the `get_return` primitive. In addition, the receiving entity defines the average connection and disconnection time periods (T_{ON}, T_{OFF}). Note that `post` and `get` operations are independent and have individual timestamps. We assume that application entities (undertaking the sender and receiver roles) enforce their semantics independently (no coordination).

DeX two-way sync. In a two-way synchronous interaction, a *client* entity posts requests using the `post_req` primitive and waits for `timeout` period to get the response via the `get_res` callback. At the server side, a thread identifier (`thr_id`) is assigned to the request procured through the `get_req` callback during the server's connected periods (i.e., T_{ON}). Subsequently, at the app-layer, the request is processed for `serve_time` and at the end of this period the *server* entity posts (`post_res`) the response associated with the same `thr_id` (unless the timeout period is reached). Finally, the message is delivered using `get_res` within the `timeout` period at the *client* side. In comparison to one-way interactions, client/server entities do not enforce their `post` and `get` semantics independently. In particular, after posting a request (`post_req`), the client is blocked to receive the response from the server using the `get_res` callback. On the other hand, requests and connection periods (T_{ON}/T_{OFF}) are initiated independently from each other.

DeX two-way async. In a two-way asynchronous interaction, a *client* entity posts requests using the `post_req` primitive with a validity period `lifetime`. The response can be received at some point later through the `get_res` callback during the client's average connected periods T_{ON}. Similarly, at the server side, the request can be procured through the `get_req` callback during the server's average connected periods T_{ON}. Subsequently, the request is processed for `serve_time` at the app-layer and at the end of this period the *server* entity posts (`post_res`) the response with a `lifetime` validity period. Finally, if the message is not expired, it is then delivered using `get_res` at the *client* side. In comparison to two-way sync interactions, client/server entities enforce their `post` and `get` semantics independently. In particular, when a client post a request

Table 1 DeX primitives per interaction type

Interaction	DeX primitives	
one way	Sender	Receiver
	`post(lifetime)`	`get(*get_return, T_ON, T_OFF)` `get_return`
two way sync	Client	Server

	`post_req(*get_res, timeout)`
	`get_res`
		`get(*get_req, T_ON, T_OFF)`
		`get_req(thr_id)`
		`post_res(thr_id)`

two way async	`post_req(lifetime)`
	`get(*get_res, T_ON, T_OFF)`
	`get_res`
		`get(*get_req, T_ON, T_OFF)`
		`get_req`
		`post_res(lifetime)`

streaming	Consumer	Producer
	`post_open(lifetime, flow_qualifier)` `get(*get_item, flow_qualifier, T_ON, T_OFF) { …get_item… }`	`get(*get_open, T_ON, T_OFF)` `get_open(flow_qualifier) { …post_item(lifetime, flow_qualifier)… }`

(`post_req`), it does not block its process to receive the response through `get_res`. Finally, requests and responses can be associated in the app-layer using identifiers that should be included in the parameters of `post` and `get` primitives.

DeX streaming. In a two-way streaming interaction, a *consumer* entity requests to establish an end-to-end connection with a producer using the `post_open` primitive. This connection is characterized using a `flow_qualifier` (i.e., a pair of <`producer`, `stream_id`>) and is valid for `lifetime` period. Multiple data items can be received for that `flow_qualifier` at some point later through the `get_item` callback during the consumer's average connected periods T_{ON}. At the producer side, the stream can be established through the `get_open` callback during the producer's average connected periods T_{ON}. Subsequently, the *producer* entity posts (`post_item`) multiple data items with a `lifetime` and the `flow_qualifier`. In comparison to two-way async interactions, the producer entity enforce posts multiple items instead of a single response.

3.3 DeX mediators

We now present how *timed DeX interactions* can be implemented by relying on the DeX API. In particular, the DeX API can be leveraged from developers to implement the `post` and `get` primitives using existing IoT protocols such as CoAP, MQTT, XMPP, etc. Then, the DeX connector model defines the composition of different primitives for implementing DeX interactions to the so called *DeX mediators* (DeXM) [3]. As depicted in Fig. 4, the mediator converts traffic data coming from a vehicle device (in JSON format through the MQTT protocol) to be received from the estimation service (in XML format through the HTTP protocol). More details on DeXM can be found in [3].

While two heterogeneous Things rely on mediators to interact with each other, the resulting end-to-end interaction is one of the DeX interactions, i.e., one-way, two-way sync/async or streaming. Therefore, we can analyze the time semantics of Things using DeX to derive properties for ensuring successful interactions. We assume that the effect of the DeX mediators on processing/transmission delays of the end-to-end interactions is negligible. We intend to extend this model by relying on [15–17] to actually consider the timing effect of the DeX mediators.

In the next section we present the time model of DeX interactions.

4 Time modeling of deX interactions

In this section, we model DeX interactions with specific emphasis on their timing behavior [4]. We propose timing models that can represent end-to-end interactions of CS,

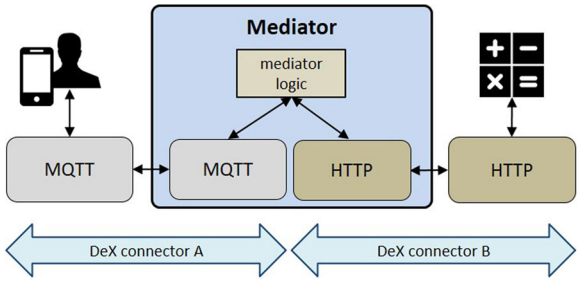

Fig. 4 Enabling *timed DeX interactions* via mediators

Table 2 Analysis parameters' and shorthand notation

Parameter(s)	Definition/Description
$t_{\texttt{post}}, t_{\texttt{get}}$	at each timestamp (t) one `post` or `get` occur.
$\delta_{\texttt{post}}, \delta_{\texttt{get}}$	the time period between two successive `post` or `get` operations.
`lifetime`	`message` availability/validity in time.
`time_on`, `time_off`	connected (`ON`) and disconnected (`OFF`) instantaneous periods for receiving messages.
`serve_time`	time needed for a request to be processed at the server side.
`timeout`	required time period to complete a request-response synchronous interaction.
$t_{\texttt{post\_req}}, t_{\texttt{get\_req}}$	at each timestamp (t) one request is sent and received, respectively.
$t_{\texttt{post\_res}}, t_{\texttt{get\_res}}$	at each timestamp (t) one response is sent and received, respectively.

PS, DS and TS systems, but also any interconnection between them through DeXM, by relying on the DeX connector model. The parameters of our timing models are depicted in Table 2.

4.1 One-way interactions

We focus here on *one-way interactions* for CS, PS, DS and TS interaction paradigms represented by DeX. In particular, our analysis considers the "steady state" behavior of PS, DS and TS interactions. In PS, subscribers have been already subscribed to receive specific events when published and they do not unsubscribe during the study period. In DS, consumers have already established a session, and in TS readers/takers accessing the tuple space properly coordinate for preventing early removal of tuples by one of the peers before all interested peers have accessed these tuples.

As shown in Fig.5, in DeX one-way the `post` primitive is used to initiate the interaction at $t_{\texttt{post}}$; also called a `post` operation. A timer is started also at $t_{\texttt{post}}$, constraining the

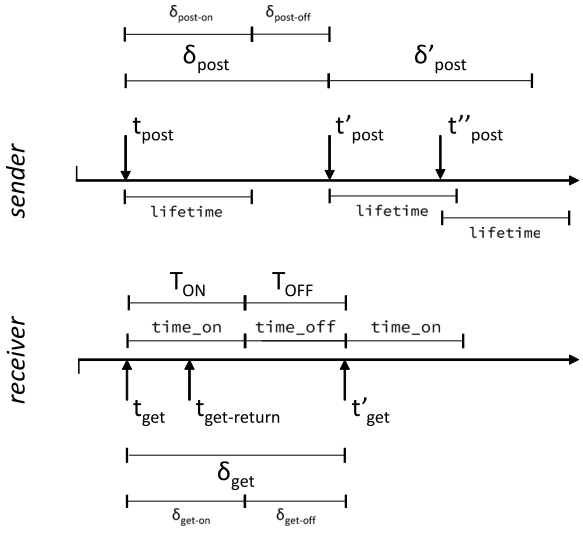

Fig. 5 Analysis of `post` and `get` δ increments for DeX one-way interactions

message availability to the `lifetime` period, also denoted by $\delta_{\texttt{post-on}}$. The period when the `lifetime` period elapses and the next `post` operation is yet to begin is denoted by $\delta_{\texttt{post-off}}$. Similarly at the receiver side, the `get` operation is initiated at $t_{\texttt{get}}$, together with a timer controlling the active period limited by the `time_on` (also denoted by $\delta_{\texttt{get-on}}$) interval. If `get` returns within the `time_on` period with valid data (not exceeding the `lifetime`), then the interaction is successful. We consider this instance also as the end of the `post` operation.

`post` operations are initiated repeatedly, with an interval rate $\delta_{\texttt{post}}$ (set as a random valued variable) between two successive `post` operations. Similarly, `get` operations are initiated repeatedly, with a random valued interval equal to $\delta_{\texttt{get}}$ between the start of two successive `time_on` periods; the interval between `time_on` and the next $t_{\texttt{get}}$ qualifies the disconnection period of receivers (`time_off` or $\delta_{\texttt{get-off}}$).

While `lifetime` and `time_on` are in general set by application/middleware designers, inter-arrival delays $\delta_{\texttt{post}}$ and $\delta_{\texttt{get}}$ are stochastic random variables dependent on multiple factors such as concurrent number of peers, network availability, user (dis)connections and so on.

Note that this model allows concurrent `post` messages; buffers of active receiving entities (including the broker and tuple space) are assumed to be infinite, hence there is no message loss due to limited buffering capacity. The message processing, transmission and queueing (due to processing and transmission of preceding messages) times inside the interaction are assumed to be negligible compared to durations of $\delta_{\texttt{post}}$ and `time_on` periods.

In particular regarding queueing, we assume that we have no heavy load effects. This means that: all posts arriving during an active period are immediately served; all posts arriving during an inactive period are immediately served at the next `time_on` period, unless they have expired before. This corresponds to a G/G/∞/∞ queueing model, where there are an infinite number of on-demand servers, hence there is no queueing. We assume that the general distribution characterizing service times incorporates the disconnections of receivers. We extend this model with actual queueing in [15–17].

Accordingly, successful one-way interactions depend on either of the disjunctive conditions:

$$t_{\texttt{get}} < t_{\texttt{post}} < t_{\texttt{get}} + \texttt{time\_on} \tag{1}$$

$$t_{\texttt{post}} < t_{\texttt{get}} < t_{\texttt{post}} + \texttt{lifetime} \tag{2}$$

meaning that a successful interaction occurs as long as a `post` and a `get` operation overlap in time. Otherwise, there is no overlapping in time between the two operations: only one of them takes place, and goes up to its maximum duration, i.e., `lifetime` for `post` and `time_on` for `get`.

Precisely:

1. If `get` occurs first, and then `post` occurs before `time_on`: the interaction is *successful*. Else, `time_on` is reached, and the `get` operation yields no interaction.
2. If `post` occur first, and then `get` occurs before `lifetime`: the interaction is *successful*. Else, `lifetime` is reached, and the interaction is a *failure*.

4.2 Two-way synchronous interactions

In CS/TS two-way sync interactions, the client sends a *request* to a server and receives the *response* from the same server within a `timeout` period. The client's processing is blocked until the interaction is complete. As depicted in Fig.6, the `post_req` operation is initiated at $t_{\texttt{post\_req}}$. A timer is started also at $t_{\texttt{post\_req}}$, constraining the request-response availability to the `timeout` period, also denoted by $\delta_{\texttt{post-req-on}}$. The period when the `timeout` period elapses and the next `post_req` operation is yet to begin is denoted by $\delta_{\texttt{post-req-off}}$. Similarly at the server side, the interval that allows to receive requests is initiated at $t_{\texttt{get}}$, together with a timer controlling the active period limited by the `time_on` (also denoted by $\delta_{\texttt{get-req-on}}$) interval. If `get_req` returns within the `time_on` period with a valid request (not exceeding the `timeout`), then after a `serve_time` interval (and only if it is still valid) the `post_res` returns the response to the client and the interaction is successful.

`post_req` operations are initiated repeatedly, with an interval rate $\delta_{\texttt{post}}$ (set as a random valued variable) between two successive `post-req` operations. Similarly, `get` operations are initiated repeatedly, with a random valued interval equal to $\delta_{\texttt{get}}$ between the start of two successive `time_on` periods; the interval between `time_on` and the next $t_{\texttt{get}}$ qualifies the disconnection period of receivers (`time_off` or $\delta_{\texttt{get-req-off}}$). `timeout` and `time_on` are in general set by application/middleware designers and inter-arrival delays $\delta_{\texttt{post}}$ and $\delta_{\texttt{get}}$ are stochastic random variables (dependent on network availability, user (dis)connections, etc).

Similar to the one-way timing model, concurrent `post` requests are allowed; (there is no message loss due to limited buffering capacity). However, once a `post_req` is active, the client blocks its operation and waits for the response during `timeout`. The request-response transmission and queueing times are assumed to be negligible compared to durations of $\delta_{\texttt{post}}$ and `time_on` periods. On the other hand, the processing of requests on the server side is defined using the `serve_time` interval.

Successful interactions depend on the following condition:

$$t_{\texttt{post\_req}} < t_{\texttt{get}} + \texttt{serve\_time} < t_{\texttt{post\_req}} + \texttt{timeout} \tag{3}$$

meaning that a successful interaction occurs as long as: *(i)* a `post_req` and a `get` operation overlap in time; and *(ii)* when there is an overlap between the `post_req` and

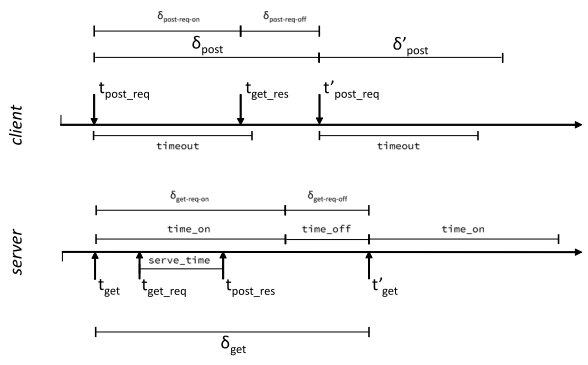

Fig. 6 Analysis of `post` and `get` δ increments for two-way synchronous interactions

the `get` operations, the request must be served before the timeout period is reached. Otherwise, there is no overlapping in time between the two operations: only one of them takes place, and goes up to its maximum duration, i.e., `timeout` for `post_req` and `time_on` for `get`.

Failed interactions occur in the following cases:

1 When `post_req` occurs, and then `get` occurs before `timeout`: the `get_req` is enforced. Else, `timeout` is reached, and the interaction results in a *failure*.
2 After enforcing the `get_req` operation, if the `get_res` occurs before `timeout`: the interaction is *successful*. Else, the `timeout` is reached due to the processing at the server side, and the interaction results in a *failure*.

4.3 Two-way asynchronous interactions

The timing behavior of two-way async and streaming interactions can be represented by relying on the time model for one-way interactions presented in subsection 4.1. Asynchronous interactions are non-blocking operations where a client sends a request to a server and then resumes its processing without waiting for a response. Based on Table 1, two-way async interactions are bidirectional where each direction is modeled using `post` and `get` primitives as follows: *(i)* a client posts a request assigned with `lifetime` using the `post_req` primitive and the server uses the `get` primitive to receive the request through the `get_req` callback; *(ii)* the server posts the response assigned with `lifetime` using the `post_res` primitive and the client uses the `get` primitive to receive the response through the `get_res` callback. Note that two-way sync interactions are bidirectional as well, however, client/server entities do not enforce their `post` and `get` primitives independently.

Accordingly, successful two-way async interactions depend on the following disjunctive conditions:

$$t_{\texttt{get}} < t_{\texttt{post\_req}} < t_{\texttt{get}} + \texttt{time\_on} \tag{4}$$

$$t_{\texttt{post\_req}} < t_{\texttt{get}} < t_{\texttt{post\_req}} + \texttt{lifetime} \tag{5}$$

$$t_{\texttt{get}} < t_{\texttt{post\_res}} < t_{\texttt{get}} + \texttt{time\_on} \tag{6}$$

$$t_{\texttt{post\_res}} < t_{\texttt{get}} < t_{\texttt{post\_res}} + \texttt{lifetime} \tag{7}$$

where any request condition can be combined with one of the response conditions for successful interaction. At $t_{\texttt{post\_req}}$ only requests are posted and `get` primitives with connectivity parameters are initiated at the server side (Eqs. 4, 5). Otherwise, at $t_{\texttt{post\_res}}$ only responses are posted and `get` primitives with connectivity parameters are initiated at the client side (Eqs. 6, 7).

4.4 Streaming interactions

Similar conditions for successful DeX streaming interactions can be derived from the conditions for successful two-way asynchronous interactions. In particular, while in asynchronous interactions a single response (message) can be followed by one request, in

streaming interactions there can be *multiple unlimited* responses. Based on Table 1, streaming interactions are bidirectional where each direction is modeled using the one-way `post` and `get` primitives as follows: *(i)* the consumer requests to open a stream using the `post_open` primitive assigned with the `lifetime` parameter. Then, the producer uses the `get` primitive to receive the open stream request through the `get_open` callback; *(ii)* the producer posts *multiple* responses assigned with `lifetime` using the `post_item` primitive and the consumer uses the `get` primitive to receive multiple responses through the `get_item` callback.

By relying on the timing models presented in this section, we can cover the various interaction types found in the IoT and represent the individual CS, PS, DS and TS paradigms, but also any heterogeneous interconnection between them, e.g., a PS subscriber interacting with a DS producer via DeX mediators.

In our future work, we aim to exploit our experience in the modeling of publish/subsbribe protocols using queueing theory [15–17] to relax the assumptions (queueing/transmission delays, message losses) of the current model. In particular, we intend to leverage features such as limited capacity, ON/OFF connectivity intervals, heterogeneous arrival rates, etc., and apply them to queueing models. The resulting queueueing networks can evaluate realistically the performance of heterogeneous IoT interactions.

5 Timed automata-based analysis

As we have analyzed the timing behaviour of DeX one-way and two-way interactions, the next step is to formally guarantee correctness of the implemented solutions. This is particularly needed in the case of IoT systems as there are varied interaction patterns with timing constraints that could be affected by incorrect parameter setting. To verify safety and reachability properties of these interactions, we make use of model checkers. Using the expressive nature of some of these models, we can include stochastic delays along with deterministic time bounds to formally model the interactions. This further allows to guarantee absence of deadlocks or livelocks within the system, which is difficult to estimate via simulations or data analysis.

In this section, we build timed automata models which represent the typical behavior of the DeX connector model for performing the timed DeX interactions described in the previous section. A timed automaton [5] is essentially a finite automaton extended with real-valued clock variables. These variables model the logical clocks in the system, which are initialized with zero when the system is started, and then increase synchronously at the same rate. Clock constraints are used to restrict the behavior of the automaton. A transition represented by an edge can be taken only when the clock values satisfy the *guard* labeled on the edge. Clocks may be reset to zero when a transition is taken. Clock constraints are also used as *invariants* at locations, which are represented by vertices: they must be satisfied at all times when the location is reached or maintained.

In order to study DeX interactions with timed automata, we make use of UPPAAL [6]. UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata. In such networks, automata synchronize via *binary synchronization channels*. For instance, with a channel declared as `chan c`, a transition of an automaton labeled with `c!` (sending action) synchronizes with the transition of another automaton labeled with `c?` (receiving action). UPPAAL makes use of computation tree logic (CTL) [18] to specify and verify temporal logic properties.

We employ the *committed location* qualifier (marked with a 'C') for some of the locations. In UPPAAL, time is not allowed to pass when the system is in a committed location; additionally, outgoing transitions from a committed location have absolute priority over normal transitions. The *urgent location* qualifier (marked with a 'U') is also used: time is not allowed to pass when the system is in an urgent location (without the priority clause of committed locations, though).

By relying on the expressive power of timed automata, we are able not only to model the timing conditions of DeX interactions, but also to introduce basic stochastic semantics regarding the behavior of peers. Using the UPPAAL model checker, we provide and verify essential properties of our timed automata model, including formal conditions for successful DeX interactions. Note that the time to verify the properties in UPPAAL is on the scale of one second maximum.

5.1 Analysis of one-Way interactions

We represent one-way DeX interactions with the connector roles *DeX sender*, *DeX receiver*, and with the corresponding *DeX one-way glue*. The two roles model the behavior expected from application components employing the connector, while the glue represents the internal logic of the connector coordinating the two roles. We detail in the following the modeling of these components.

Figure 7 shows the *sender* behavior. Typically, a sender entity repeatedly emits a `post!` action (`message`) to the *glue* without receiving any feedback about the end (successful or not) of the `post` operation. We have enhanced (and at the same time constrained) the sender's behavior with a number of features. The committed locations `post_event` (`post!` sent to the glue) and `post_end_event` (`post_end?` received from the glue) have been introduced to detect the corresponding events. Upon these events, the automaton oscillates between the `post_on` and `post_off` locations, which correspond to the $\delta_{\texttt{post-on}}$ and $\delta_{\texttt{post-off}}$ intervals presented in Fig. 5. `delta_post` is a clock that controls the $\delta_{\texttt{post}}$ interval between two successive `post` operations. `delta_post` is reset upon a new `post` operation and set to `lifetime` at the end of this operation (note that the `post_init` location and its outgoing transition serve to initialize `delta_post` at the beginning of the sender's execution – this unifies verification also for the very first `post` operation). The invariant condition `delta_post<=max_delta_post` (where

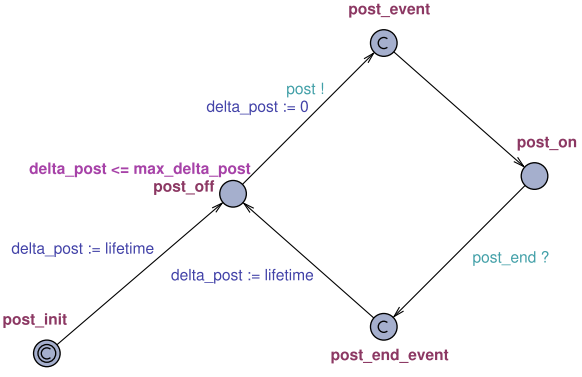

Fig. 7 DeX sender automaton

max_delta_post is a constant) at the post_off location ensures that a new post operation will be initiated before the identified boundary.

This setup results in at most one post operation active at a time. This post remains active ($\delta_{\texttt{post-on}}$ interval) for lifetime interval (and then it expires) or less than lifetime interval (in case of successful interaction). In both cases, we set delta_post to lifetime at the end of the post operation (this enables verification, since we can not capture absolute times in UPPAAL). Hence, the immediately following $\delta_{\texttt{post-off}}$ interval will last a stochastic time uniformly distributed in the interval [lifetime, max_delta_post]. With regard to the one-way timing model of Section 4, we opted here for restraining concurrency of post operations for simplifying the architecture of the glue. The present model (sender, receiver and one-way glue) can be compared to one of the infinite on-demand servers of the G/G/∞/∞ model of Section 4. Nevertheless, this model is sufficient for verifying Conditions (1) and (2) for successful DeX interactions. These conditions relate any post operation with an overlapping get operation; possible concurrency of post operations has no effect on this. Moreover, in the following sections we prove that these conditions are independent of the probability distributions characterizing the sender and receiver's stochastic behavior.

Figure 8 shows the *receiver* behavior. Typically, a receiver entity repeatedly emits a get! action to the glue, with at most one get operation active at a time. The duration of the get operation is controlled by the receiver with a local time_on; upon the time_on, a get_end! action is sent to the glue. Before reaching the time_on, multiple messages (posted by senders) may be delivered to the receiver by the glue, each with a get_return? action. We have enhanced the receiver's behavior with similar features as for the sender. Hence, we capture the events and time intervals presented in Fig. 5 with the get_event, get_end_event, get_on, get_off locations, as well as with the delta_get clock and the invariant conditions delta_get<=time_on (at get_on) and delta_get<=max_delta_get (at get_off). This setup results in a succession of $\delta_{\texttt{get-on}}$ and $\delta_{\texttt{get-off}}$ intervals, with the former lasting time_on time and the latter lasting a stochastic time uniformly distributed in the interval [time_on, max_delta_get]. We have additionally introduced the committed location no_trans, which, together with the Boolean variable get_ret, helps detecting whether the whole time_on period elapsed with no interaction performed or at least one message was received.

The *glue* one-way automaton is shown in Fig. 9. It determines the synchronization of the incoming post? and get? operations. A successful synchronization between such

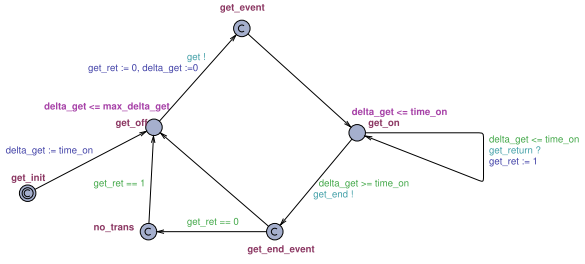

Fig. 8 DeX receiver automaton

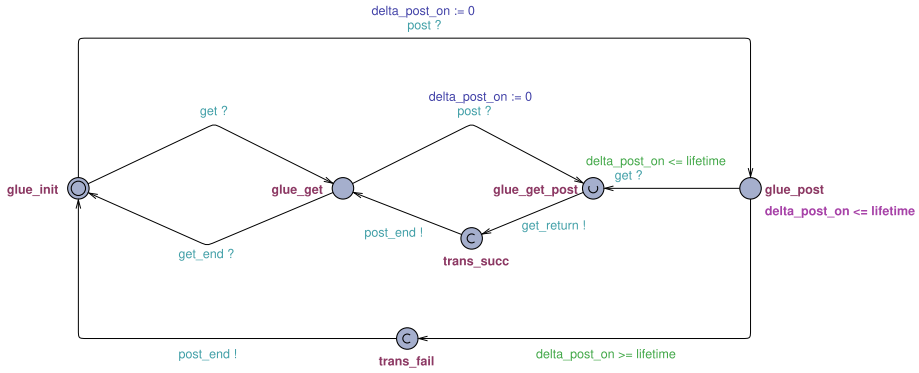

Fig. 9 DeX glue one-way automaton

operations leads to a successful interaction, which is represented in the automaton by the `trans_succ` location. Note that the timing constraints specified in Section 4 regarding the lifetime of posted `messages` have been applied here with the additional clock `delta_post_on` employed to guard transitions dependent on the `lifetime` period. Two ways for reaching the `trans_succ` location are considered:

- If the `get?` operation occurs from the initial location (leading to location `glue_get`), a consequent `post?` operation results in a `get_return! message` and eventually the successful interaction location `trans_succ` (Eq. 1). At the same time, the sender is notified of the end of the `post` operation with `post_end!`. Note that we employ the *urgent location* qualifier for `glue_get_post`; thus, the glue completes instantly the successful interaction and is ready for a new one. At the `glue_get` location, if the `get_end?` action is received from the receiver automaton (suggesting `delta_get >= timeout`), the glue is reset to the initial location `glue_init`.
- If the `post?` operation occurs initially (leading to location `glue_post`), a `get?` operation before the constraint `delta_post_on <= lifetime` results again in a successful interaction (Eq. 2). Exceeding the `lifetime` period without any `get?` results in location `trans_fail`, and the automaton returns to its initial location `glue_init`, notifying at the same time the sender with `post_end!`. This is done without any delay, thanks to the invariant `delta_post_on <= lifetime` at the `glue_post` location.

5.1.1 Verification of properties

We verify reachability and safety properties of the combined automata *DeX sender*, *DeX receiver* and *DeX glue one-way*, by using the model checker of UPPAAL. A reachability property, specified in Uppaal as `E<>`φ, expresses that, starting at the initial state, a path exists such that the condition φ is eventually satisfied along that path. A safety property, specified in UPPAAL as `A[]`φ, expresses that the condition φ invariantly holds in all reachable states.

Sender Automaton. We verify a set of reachability and safety properties that characterize the timings of the sender's stochastic behavior.

```
A[] sender.post_event imply delta_post==0                                    (8)

A[] sender.post_on imply delta_post<=lifetime                                (9)

A[] sender.post_off imply (delta_post>=lifetime and
                           delta_post<=max_delta_post)                       (10)

E<> sender.post_end_event and delta_post< lifetime                           (11)
```

Equation 8 states that `post` events occur at time 0 captured by the `delta_post` clock. Equation 9 and 11 together state that [0, `lifetime`] is the maximum interval in which a `post` operation is active; nevertheless, the operation can end before `lifetime` is reached. Equation 10 states that [`lifetime`, `max_delta_post`] is the maximum interval in which there is no active `post` operation. This confirms the fact that we artificially "advance time" to `lifetime` at the end of the `post` operation.

Receiver Automaton. We verify similar properties that characterize the timings of the receiver's stochastic behavior.

```
A[] receiver.get_event imply delta_get==0                                    (12)

A[] receiver.get_on imply delta_get<=time_on                                 (13)

A[] receiver.get_off imply (delta_get>=time_on and
                            delta_get<=max_delta_get)                        (14)

A[] receiver.get_end_event imply delta_get==time_on                          (15)
```

Hence, Eq. 12 states that `get` events occur at time 0 captured by the `delta_get` clock. Equation 13 and 15 together state that a `get` operation precisely and invariantly terminates at the end of the [0, `time_on`] interval. Equation 14 states that [`time_on`, `max_delta_get`] is the maximum interval in which there is no active `get` operation.

Glue one-way Automaton. We verify conditions for successful interactions using the glue automaton.

```
A[] glue.trans_succ imply (sender.post_on and receiver.get_on
                           and (delta_post==0 or delta_get==0))               (16)
```

In addition to the reachability property (`E <> glue.trans_succ`), we verify the safety property in Eq. 16. According to this, a successful interaction event implies that while a `post` operation is active a `get` event occurs, or while a `get` operation is active a `post` event occurs.

```
A[] glue.trans_fail imply (sender.post_on and receiver.get_off
     and delta_post==lifetime and delta_get-time_on>=lifetime)                (17)
```

We verify both the reachability property (`E <> glue.trans_fail`) and the safety property in Eq. 17. A failed interaction event means that `lifetime` is reached for an active `post` operation and no `get` operation is active. Additionally, the ongoing inactive `get` interval entirely includes the terminating active `post` interval. With regard to the stochastic `post` and `get` processes of our specific setting, we explicitly checked that if the

condition `max_delta_get-time_on>=lifetime` does not hold for the given values of the included constants, then the reachability property `E<> glue.trans_fail` is indeed not satisfied.

```
A[] receiver.no_trans imply (receiver.get_on and sender.post_off
        and delta_get==time_on and delta_post-lifetime>=time_on)
```
(18)

We verify both the reachability property (`E <> receiver.no_trans`) and the safety property in Eq. 18. Symmetrically to Eq. 17, a no-interaction event implies that `time_on` is reached for an active `get` operation and no `post` operation is active. Additionally, the ongoing inactive `post` interval entirely includes the terminating active `get` interval. Similarly to Eq. 17, we check that if this safety property is not satisfied, then the state `receiver.no_trans` is indeed not reachable.

Observing Eqs. 16, 17, 18, we see (as intuitively expected) that successful, failed and no-interactions are determined by the durations and relative positions in time of the $\delta_{\texttt{post-on}}$, $\delta_{\texttt{post-off}}$, $\delta_{\texttt{get-on}}$ and $\delta_{\texttt{get-off}}$ intervals. These depend on the deterministic parameter constants `lifetime`, `time_on` and on the stochastic parameters $\delta_{\texttt{post}}$ and $\delta_{\texttt{get}}$. It is also worth nothing that Eqs. 16, 17, 18 are expressed in a general way, independently of the specific `post` and `get` stochastic processes. For example, Eq. 17 states that, to have a failed interaction, a lifetime period must be lower than the `time_off` period. But `time_off` is probabilistic. Therefore, to avoid failed interactions, a system designer can tune the system by changing (or trying to affect) these parameters accordingly, while they can employ any probability distribution for the disconnection parameter. Similarly, Eq. 18 provides the developer with hints of how to possibly avoid no-interactions. Hence, the analysis results of this section provide general formal conditions for successful DeX interactions and their reliance on observable and potentially tunable system and environment parameters. Using these results, we perform experiments to quantify the effect of varying these parameters for successful interactions in Section 6.

5.2 Analysis of two-Way synchronous interactions

We represent two-way synchronous DeX interactions with the connector roles *DeX client*, *DeX server*, and *DeX two-way sync glue*. We detail in the following the modeling of these components.

Figure 10 shows the *client* behavior. Typically, a client emits a `post_req` (*request*) to the *glue* and waits for `timeout` to receive the `get_res` (*response*). The committed location `post_req_sent` is introduced to detect the event of sending a request (`post_req!`) to the glue. Upon such an event, the automaton stays on the `post_req_on` location to either receive the response or until the `timeout` expires, which corresponds to the $\delta_{\texttt{post-req-on}}$ interval presented in Fig. 6. Upon the timeout expiration or the `get_res` reception, the automaton stays in the `post_req_off` for $\delta_{\texttt{post-req-off}}$ time period.

`delta_post` is a clock that controls the $\delta_{\texttt{post}}$ interval between two successive `post_req` operations. `delta_post` is reset upon a new `post_req` operation and set to `timeout` upon a `get_res?` (prior to the `timeout` expiration). On the other hand, when the timeout period is reached, the `delta_post` clock is already set to timeout on the `post_req_off` location. We initialize `delta_post` at the beginning of the client's execution (`post_init` location). The invariant condition

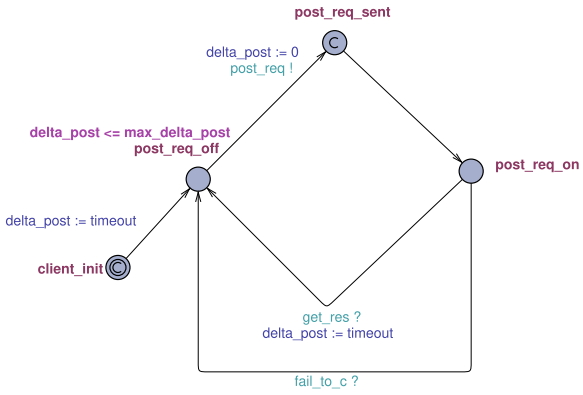

Fig. 10 DeX client automaton

`delta_post<=max_delta_post` (where `max_delta_post` is a constant) at the `post_req_off` location ensures that a new `post_req` operation will be initiated before the identified boundary.

Based on the above setup, the client sends at most one `post_req` operation active at a time. This request remains active ($\delta_{\texttt{post-req-on}}$ interval) for `timeout` period (and then it expires) or less than `timeout` period (in case of successful interaction). In both cases, `delta_post` equals `timeout` at the end of the `post_req` operation. Hence, the immediately following $\delta_{\texttt{post-off}}$ interval will last a stochastic time uniformly distributed in the interval [`timeout`, `max_delta_post`]. Such a model is sufficient for verifying the condition (Eq. 3) for successful DeX two-way sync interactions. This condition relates any `post_req` operation with an overlapping `get` operation, by taking also into account the deterministic parameter `serve_time` at the server side.

Figure 11 shows the *server* behavior. Typically, a server entity repeatedly becomes online (location `get_on`) to receive requests from the glue. Thus, the server automaton oscillates between the locations `get_off` and `get_on`. The `get_event` committed location is used to detect the online status of the server. It is worth noting that the server entity operates independently from the glue – i.e., it does not notify the glue when changing between the `get_on` and `get_off` locations. The automaton stays on the `get_on`

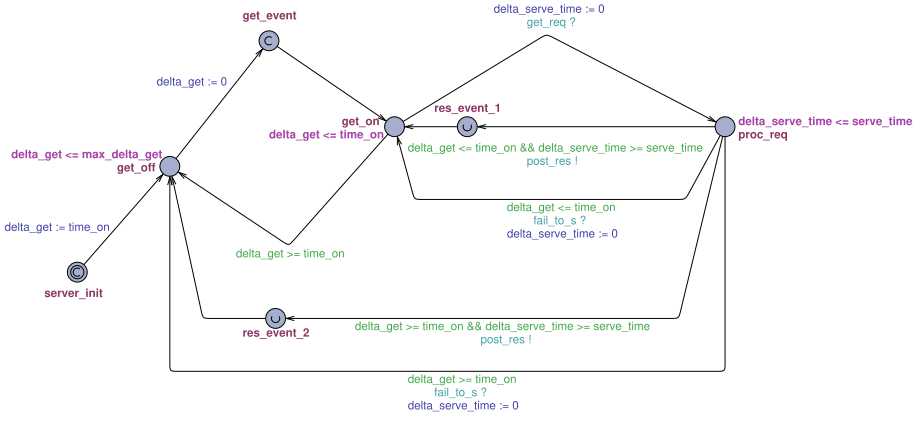

Fig. 11 DeX server automaton

location for a specific interval, which is controlled by the server with a local `time_on`; upon the `time_on`, the automaton returns to the `get_off` location. Similar to the client entity, the `delta_get` clock is used to measure the `time_on` interval and switch between the two locations (`get_on` and `get_off`). Furthermore, the invariant conditions `delta_get<=time_on` (at `get_on`) and `delta_get<=max_delta_get` (at `get_off`) guarantee the correct operation of our automaton.

Before reaching the `time_on`, multiple `requests` (posted by clients) may be delivered to the server by the glue, each with a `get_req?` action. Upon a `get_req?`, the automaton stays in the `proc_req` location for `serve_time` interval, which corresponds to the necessary time period for processing a request. We use the *urgent* `res_event_1` and `res_event_2` locations to detect successful responses through the `post_res!` action. Particularly, the `res_event_1` location is reached only if the server is still online (`delta_get<=time_on`). However, while being in location `proc_req`, the server entity may become offline. For such case, the `res_event_2` location is reached after serving the request (because of the invariant `delta_serve_time<=serve_time`), and then the automaton returns to the `get_off` location. Finally, the automaton returns to `get_on` or `get_off` locations upon a `fail_to_s?` action received by the glue, which corresponds to the request (`post_req`) expiration due to the `timeout` period.

This setup results in a succession of $\delta_{\texttt{get-req-on}}$ and $\delta_{\texttt{get-req-off}}$ intervals (see Fig. 6), with the former lasting `time_on` time and the latter lasting a stochastic time uniformly distributed in the interval [`time_on`, `max_delta_get`].

The *glue* two-way sync automaton is shown in Fig. 12. It determines the synchronization of the incoming (`post_req?` and `post_res?`) and outgoing (`get_req!`) operations. A successful synchronization between such operations leads to a successful interaction, which is represented in the automaton by the `trans_succ` location. Note that the timing constraints specified in Section 4 regarding the `timeout` of sent requests have been applied here with the additional clock `delta_req_on` employed to guard transitions dependent on the `timeout` period.

The `trans_succ` location is reached through the following operations: if the `post_req?` operation occurs from the initial location and the invariant `delta_req_on <= timeout` is satisfied, a consequent `get_req!` request is sent to

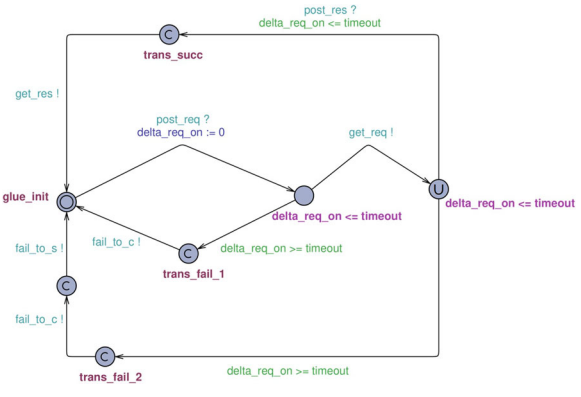

Fig. 12 DeX glue two-way synchronous automaton

the server (if the server automaton is on the location `get_on`). While the request is processed on the server side, the glue automaton waits for the reply. After the specified `server_time` a `post_res?` operation occurs to the glue and eventually the successful interaction location `trans_succ` (the Eq. 3 is satisfied). At the same time, the client is notified of the end of the `post_req` operation with `get_res!`. Note that we employ the `get_req` channel as `urgent`. In this way, upon a `post_req?` and if the server is online, the `get_req!` action occurs instantly, without any delay as indicated by the invariant `delta_req_on<=timeout`.

With regard to the `timeout`, `time_on` and `serve_time` parameters, we identify failed interactions in the glue trough the `trans_fail_1` and `trans_fail_2`. Two ways for reaching the fail locations are considered:

- If the `post_req?` operation occurs from the initial location and the server automaton is offline (stays on the `get_off` location) for a time period that leads to the timeout expiration (`delta_req_on>=timeout`), the `trans_fail_1` location is reached. At the same time, the client is notified with `fail_to_c!` in order to move at the `post_req_off` location.
- If the `post_req?` operation occurs from the initial location and the server automaton is online (stays on the `get_on` location), a consequent `get_req!` request is sent to the server. While the request is processed for `serve_time`, the timeout period may expire (`delta_req_on>=timeout`) and the `trans_fail_2` location is reached. At the same time, the client is notified with `fail_to_c!` to move at the `post_req_off` location, and the server is notified with `fail_to_s!` to move either to `get_on` or to `get_off` locations, depending of the `delta_get` clock.

5.2.1 Verification of properties

We verify reachability (`E<>`φ) and safety (`A[]`φ) properties of the combined automata *DeX client*, *DeX server* and *DeX two-way sync glue*, by using the model checker of UPPAAL.

Client Automaton. We verify a set of safety properties that characterize the timings of the client's stochastic behavior.

```
A[] client.post_req_sent imply delta_post==0                          (19)
```

```
A[] client.post_req_sent imply delta_post<=timeout                    (20)
```

```
A[] client.post_req_off imply (delta_post>=timeout and
                               delta_post<=max_delta_post)            (21)
```

Equation 19 states that `post_req` events occur at time 0 captured by the `delta_post` clock. Equation 20 states that $[0, \texttt{timeout}]$ is the maximum interval in which a `post_req` operation is active, nevertheless, the operation can end before `timeout` is reached. Equation 21 states that $[\texttt{timeout}, \texttt{max\_delta\_post}]$ is the maximum interval in which there is no active `post_req` operation. Similar to the *DeX sender* automaton, we artificially "advance time" to `timeout` at the end of the `post_req` operation.

Server Automaton. We verify similar properties that characterize the timings of the server's stochastic behavior.

```
A[] server.get_event imply delta_get==0                                    (22)
```

```
A[] server.get_on imply delta_get<=time_on                                 (23)
```

```
A[] server.get_off imply (delta_get>=time_on and
                          delta_get<=max_delta_get)                        (24)
```

Equation 22 states that at the beginning of the server's online period, the automaton passes from the location `get_event` at time 0 captured by the `delta_get` clock. Equation 23 states that the server stays online (at the location `get_on`) at least for `time_on` interval. Equation 24 states that `[time_on,max_delta_get]` is the maximum interval in which the server is offline (at the location `get_off`).

Glue two-way sync Automaton. Finally, we verify conditions for successful interactions using the glue automaton.

```
A[] glue.trans_succ imply (client.post_req_on and delta_post<=timeout
                    and (server.res_event_1 or server.res_event_2)
                          and delta_get<=time_on+serve_time)                (25)
```

We verify both the reachability property (`E <> glue.trans_succ`) and the safety property in Eq. 25. According to this, a successful interaction event implies that while a `post_req` operation is active the `timeout` period is not reached. Additionally on the server side, one of the committed locations `res_event_1` or `res_event_2` is active and the condition `delta_get<=time_on+serve_time` holds.

```
A[] glue.trans_fail_1 imply (client.post_req_on and delta_post==timeout
                    and ((server.get_off and delta_get-time_on>=timeout)
                          or (server.get_on and delta_get==0)))             (26)
```

We verify both the reachability property (`E <> glue.trans_fail_1`) and the safety property in Eq. 26. A failed interaction event means that `timeout` is reached for an active `post_req` operation. Additionally, the request can not reach the server either because it is offline (`get_off` location) for time period greater of timeout (`delta_get-time_on>=timeout`), or due to the fact that the server automaton moved on to location `get_on` and at the same time the `timeout` period is reached.

```
A[] glue.trans_fail_2 imply (client.post_req_on and delta_post==timeout
                    and server.proc_req and delta_get<=serve_time)          (27)
```

Upon an interaction if the above condition is not verified, it means that the request is processed at the server side. However, an additional failure can occur in location `trans_fail_2` while the request is processed. In addition to the reachability property (`E<> glue.trans_fail_2`), we verify the safety property in Eq. 27. Such a failed interaction event means that `timeout` is reached for an active `post_req` operation. Additionally, the request is processed in location `proc_req` since the condition `delta_get<=serve_time` is valid.

Equations 25, 26 and 27 provide us with general formal conditions which can be utilized by system designers to tune timing parameters such as `timeout`, `time_on` and `serve_time` and achieve successful interactions.

5.3 Analysis of two-Way asynchronous and streaming interactions

Based on subsections 4.3 and 4.4, the time models for one-way interactions can be leveraged to model the timing behavior of two-way async and streaming interactions. Similarly, the Timed Automata models provided in this section be can be leveraged to derive general formal conditions for successful two-way async and streaming interactions. In particular, we leverage the *glue one-way automaton* shown in Fig. 9, to derive conditions similar to Eqs. 16, 17, 18 using two-async and streaming DeX operations based on Table 1. For example, in two-way async interactions, we can verify the safety property for successful request transmissions as follows:

```
A[] glue.trans_succ imply (client.post_req_on and server.get_on
                        and (delta_post_req==0 or delta_get==0))
```
(28)

According to the above, a successful request transmission implies that while a `post_req` operation is active a `get` event occurs at the server side, or while a `get` operation is active at the server side, a `post_req` event occurs. Similarly, formal conditions for successful/failed requests, responses, open stream requests and delivery of stream items can be derived. Such conditions can be leveraged by system designers to tune timing parameters.

5.4 Summary of verification outputs

In this section, we have provided a detailed view of timed automata modeling, verification and parameter tuning of one-way, two-way and streaming interactions. Having a unified framework for verification allows us to study the effect of timing delays, lifetime parameters and success rates of transmissions. This process does not need to be repeated by users of DeXM within unique deployment scenarios. The verified properties demonstrate that the `get` and `post` operations can be managed in a safe way via the timing constraints provided. While we have concentrated on safety and reachability properties composing the ordering of event operations, the models may be reused to verify other associated properties. This formalizes the notion of successful interactions within DeXM.

6 Simulation-based analysis

In this section, we provide results of simulations of DeX one-way interactions with varied `lifetime` and `time_on` periods. We demonstrate that varying these periods has a significant effect on the rate of successful interactions. Furthermore, the trade-off involved between delivery success rates and response times (depending on `lifetime`/`time_on` periods) is evaluated. Finally, we validate our simulation-based analysis by using the DeXM framework which provides implementations of the DeX interactions through real middleware protocols.

Table 3 provides an overview of the experimental settings used in the simulations. These settings are derived from working with real deployments of Middleware such as Java Messaging Service and DPWS.

Table 3 Experimental settings

Parameter(s)	Definition/Description	Experimental Settings
$t_{\texttt{post}}$, $t_{\texttt{get}}$	at each timestamp (t) one `post` or `get` occur.	Poisson arrival rate for $t_{\texttt{post}}$; Exponential inter-arrival time for $t_{\texttt{get}}$.
$\delta_{\texttt{post}}$, $\delta_{\texttt{get}}$	the time period between two successive `post` or `get` operations.	Exponential distribution for $\delta_{\texttt{post}}$ with mean 10 sec.; Exponential distribution for $\delta_{\texttt{get}}$ with mean between 10 sec. to 40 sec.
`lifetime`	`message` availability/validity in time.	0 sec. to 40 sec.
`time_on`, `time_off`	connected (`ON`) and disconnected (`OFF`) instantaneous periods for receiving messages.	`time_on` between 10 sec. to 60 sec.
`timeout`	required time period to complete a request-response synchronous interaction.	Derived from the measurements.

6.1 Delivery success rates

In order to test the effect of varying `lifetime` and `time_on` periods on interaction success rates, we perform simulations over the timing analysis one-way model described in Section 4. *Poisson* arrival rates are assumed for subsequent t_{post} instances (hence, $\delta_{\texttt{post}}$ follows the corresponding exponential distribution). Each message is valid for a deterministic `lifetime` period and then discarded. Similarly, there are *exponential* intervals between subsequent $t_{\texttt{get}}$ periods ($\delta_{\texttt{get}}$ follows this distribution). The receiver entity is active for a deterministic `time_on` period and can disconnect for random valued intervals. Applying the one-way timing model in Section 4, the simulation enables concurrent posts with no-queueing. As the arrivals follow a *Poisson* process, this simulates an M/G/∞/∞ queueing model.

The simulations done in *Scilab*[1] analyze the effect of varying `lifetime` and `time_on` periods on DeXM interactions. We set $\delta_{\texttt{post}}$ between subsequent `post` messages correspond to `post` operations presented in 5.1) to have a mean of 10 sec. Note that an IoT designer can select different parameters for the subsequent `post` messages. Nevertheless, in this case the `lifetime` and `time_on` periods should carefully be selected in order to highlight their effect on system tuning. The `get` messages correspond to `get` operations presented in 5.1) are simulated with varying exponential active periods ($\delta_{\texttt{get}}$). This procedure was run for 10,000 $t_{\texttt{get}}$ periods (enough runs to converge to the average periods selected) to collect interaction statistics with 95% confidence interval setting; we applied the formal conditions of subsection 5.1 to control the rates of successful interactions. As depicted in Fig. 9, a successful synchronization between `post` and `get` operations leads to a successful DeX interaction, which is represented in the automaton by the *trans_succ* location. The experiments were conducted over a Linux machine with Intel Core i7, 16 Gb RAM.

The rates of successful interactions are shown in Fig. 13 for various values of `lifetime`, `time_on` and δ_{get} periods. As expected, increasing `time_on` periods for individual `lifetime` values improves the success rate. However, notice that the success rate is severely bounded by `lifetime` periods. For instance, when the `lifetime` period is very low (0 sec), the success rate, even at higher `time_on` intervals, remains bound at around 70% for $\delta_{\texttt{get}}$ with mean 40 sec. Such behavior represents time/space coupled CS interactions, where each message is received immediately by the client (we assume

[1] http://www.scilab.org

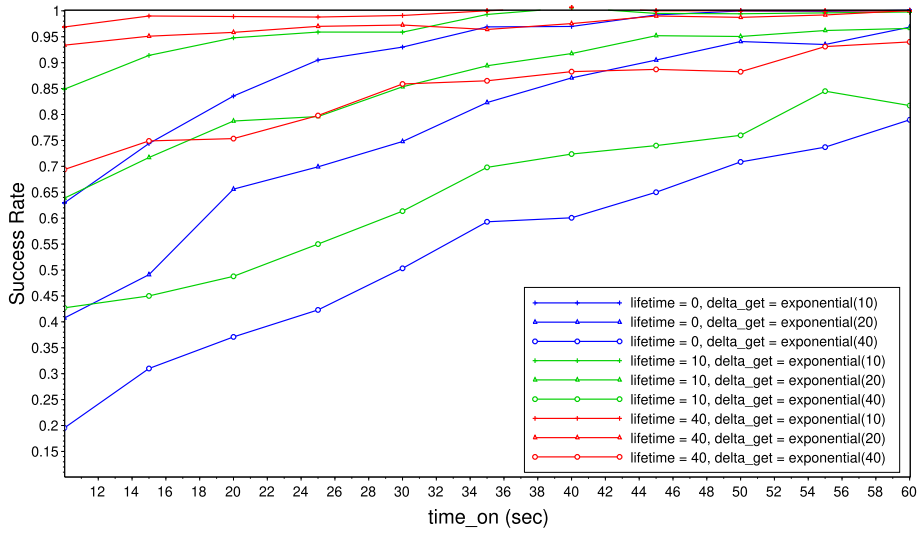

Fig. 13 Delivery success rates with varying `time_on` and `lifetime` periods

that the transmission delay of the underlying network delay is negligible). Reducing `get` disconnection intervals (by properly setting $\delta_{\texttt{get}}$ and `time_on`) produces a significant improvement in the success rate, especially for the CS case. For the other interaction paradigms (PS/TS employing an intermediate middleware node), where the `lifetime` period can be varied: a higher `lifetime` period combined with higher `time_on` or lower $\delta_{\texttt{get}}$ intervals would guarantee better success rates.

6.2 Response time vs. delivery success rate

In order to study the trade-off between end-to-end response time and delivery success rate, we present cumulative response time distributions for interactions in Fig. 14. Note that we assume that all posts arriving during an active `get` period are immediately served;

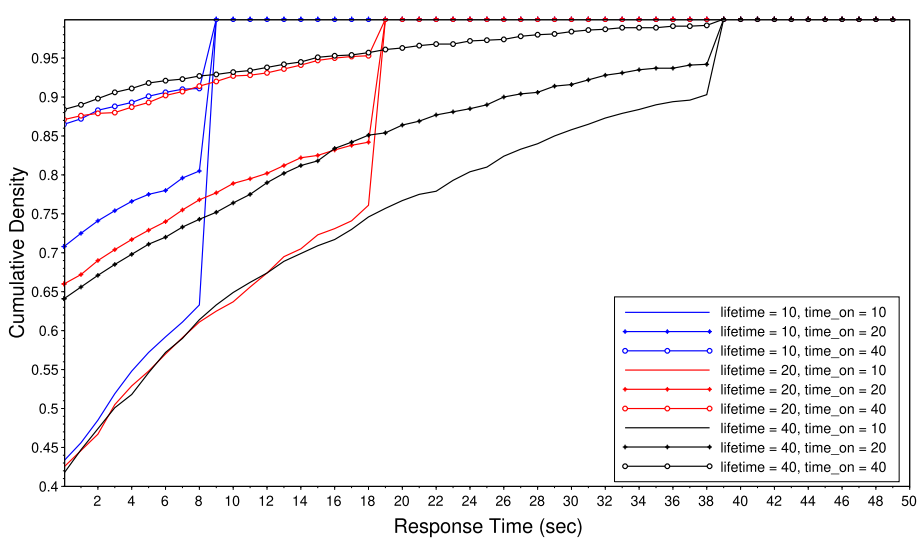

Fig. 14 Response time distributions for interactions with varying `time_on` and `lifetime` periods

all posts arriving during an inactive `get` period are immediately served at the next active period, unless they have expired before. All failed transactions are pegged to the value: `lifetime`.

We set $\delta_{\texttt{post}} = Poisson(10)$ sec and $\delta_{\texttt{get}} = Exponential(20)$ sec for all simulated cases. From Fig. 14, lower `lifetime` periods produce markedly improved response times. For instance, with `lifetime` = 10 sec, `time_on` = 20 sec, all interactions complete within 10 sec. Comparing this to Fig. 13, the success rate with these settings is 78%. Changing to `lifetime` = 40 sec, `time_on` = 20 sec, we get a success rate of 95%, but with increased response time. So, with higher levels of `lifetime` periods (typically PS/TS), we notice high success rates, but also higher response time. While individual success rates and response time values depend also on the network/middleware efficiency, our analysis provides general guidelines for setting the `lifetime` and `time_on` periods to ensure successful interactions.

6.3 TIM system tuning

Our fine-grained timing analysis can be employed to properly configure the TIM system. Accordingly, `vehicle-devices` and `fixed-sensors` emit `posts` carrying traffic-related messages with a mean arrival rate of 1 event every 10 min. To guarantee the freshness of provided information, notifications are maintained by the system for a `lifetime` period of 10 min. We assume that `smartphone-users` access the system every 20 min on average to receive up-to-date transport information on their hand-held devices. They stay connected for a `time_on` period and then disconnect, also for resource saving purposes. Actual connection/disconnection behavior is based on the user's profile. By relying on our statistical analysis, an application designer may configure the `time_on` period of user access to 10 min. Using scaled values from Figs. 13 and 14, this guarantees that the user will receive on average 65% of the posted notifications, within at most 8 min of response time with a probability of 0.63. If these values are insufficient and the designer re-configures the `time_on` to 20 min, this guarantees that now the user will receive on average 80% of the posted notifications within at most 4 min of response time with a probability of 0.77.

6.4 Comparison with deXM implementation

In order to validate the simulations performed in Section 6.1, we implement realistic interactions using the DeXM framework. As depicted in Fig. 15, we create two pairs (sender-receiver) of mockup mobile IoT devices using different middleware protocols. In particular, we use two middleware implementations: *(i)* for `lifetime` = 0 transactions, the DPWS[2] CS middleware provides an API to set a sender and a receiver interacting with each other directly; and *(ii)* for (`lifetime` > 0) transactions, the JMS[3] PS middleware provides an API to set a sender, a receiver, and the intermediate entity through which they interact. Applying the same settings as in Section 6.1, senders and receivers perform operations based on probability distributions (i.e., exponential $\delta_{\texttt{post}}$ with mean of 10 sec and $\delta_{\texttt{get}}$ with various mean periods). At the intermediate entity we set various `lifetime` periods, using the JMS API. All the interactions are performed using an Intel Xeon W3550e 3.08 GHz × 4 (7.8GB RAM) under a *Linux Mint* OS. Note that

[2] http://ws4d.e-technik.uni-rostock.de/jmeds
[3] http://activemq.apache.org

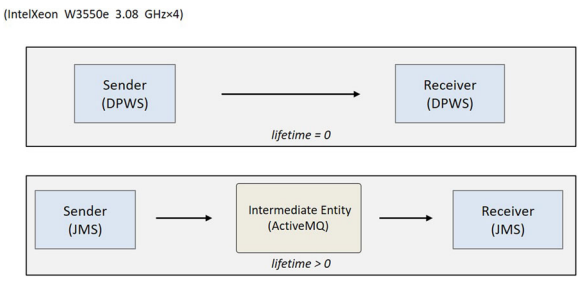

Fig. 15 DeXM experimental setup

in these DeXM implementation settings, we have concurrent `posts` and queueing. This corresponds to an M/G/1/∞ queueing model; however, the queueing time of data due to processing of preceding data is negligible in our specific settings.

For getting reliable results, the mean values of $\delta_{\texttt{post}}$ and $\delta_{\texttt{get}}$ intervals are expected to be close to the expected mean values. To do so, we create sufficient number of `post` operations and `get` connections/disconnections by running each experiment for at least 2 hours. In Table 4, we compare the results of simulated and measured success rates for `time_on` $=$ 20 sec, $\delta_{\texttt{post}} = \mathit{Poisson}(10)$ sec, `lifetime` $= 0, 10, 40$ sec and various distributions for $\delta_{\texttt{get}}$. The absolute deviation between the two is no more than 10%. This deviation may be attributed to implementation factors such as buffering at each entity (sender, receiver, intermediate entity) which may affect the success rates. As this deviation is not too high, it allows developers to rely on our simulation model to tune the system.

7 Related work

IoT devices exchange data determined by functional semantics such as space, time and synchronization coupling. In the mobile IoT, the data exchange is highly dependent on *time semantics* such as data availability/validity, intermittent connectivity and app-layer processing time. Consequently, investigating generic evaluation techniques of such system semantics is crucial.

In [19], the authors express the intermittent WiFi availability for a mobile user using 2-D Markov chains. This is a complex and tedious procedure. Extending this approach to mobile peers of middleware systems (e.g., publish/subscribe) can be even more complicated, due to the combined modeling of network-level and application-level time semantics. Alternatively, *Queueing Network (QNs)* and *Performance Petri Nets (PPNs)* are both "high level" flexible techniques for describing (primarily) Markov models which can be used for constructing performance metrics about computer systems and, subsequently, middleware systems [20]. The notation used to describe the models enables the user

Table 4 Simulated vs. measured delivery success rates

`lifetime` (s)	$\delta_{\texttt{get}}$ (s)	Simulation	Measurement
0	`exponential(20)`	0.65	0.717
0	`exponential(40)`	0.35	0.42
10	`exponential(20)`	0.75	0.778
10	`exponential(40)`	0.48	0.554
40	`exponential(20)`	0.93	0.91
40	`exponential(40)`	0.75	0.81

to develop and explore a large design space rapidly. Along the dimensions of *expressive power* and *solution efficiency*, PPNs enjoy an advantage over QNs in representing synchronization (parallel systems) and are probably best suited for design purposes. A closely related work is [13], where formal analysis (using colored Petri-Nets) of various types of time synchronization in distributed middleware architectures has been performed. On the other hand QNs provide convenient primitives for constructing models, guarantee that are well-formed (i.e., stable, deadlock-free, etc), and can be solved efficiently. Work done by Kattepur and Nambiar [21] makes use of QNs to estimate performance of Web applications using algorithms such as *Mean Value Analysis* (MVA).

In comparison to the above approaches that are used to model and analyze queueing and synchronization phenomena of systems, we model DeX interactions (which unify heterogeneous underlying mobile IoT interactions) by focusing on the high-level timing semantics of mobile peers. This combines network-level and application-level semantics, while assuming that the message processing, transmission and queueing times inside an interaction are negligible compared to the timing behaviors of mobile peers. Hence, we opt for employing statistical modeling, simulation and subsequent analysis for deriving general performance metrics for heterogeneous mobile IoT systems.

On the other hand, timed automata [5] can be used to model and analyze the formal timing behavior of computer systems, e.g., real-time systems or protocols. They have been applied to a variety of real time system models to ensure accurate behavior under timed guards. Such models enable checking both safety and liveness properties and they have been particularly developed and studied over the last years. Model checkers such as UPPAAL [6], PRISM [22] and SBIP [23] have been proposed for analyzing timed and probabilistic properties of systems. Timed automata are used in [24] for studying fault tolerant behavior (safety, bounded liveness) in distributed asynchronous real time systems. Furthermore, in [25], a hierarchical timed automata based approach is proposed to model and analyze the dynamic software evolution of service oriented systems. Both functional evolution (with structural changes of the software architecture) and non-functional evolution (with parameter changes) are considered. Hierarchical timed automaton (HTA) introduces a refinement function to map a composite location to an underlying set of automata, and hence can describe the hierarchical structure of service composition.

In [26], the transmission channels of publish/subscribe middleware as well as overlying application components are modeled using probabilistic timed automata for verifying properties of the supported interactions with the PRISM probabilistic model checker. The same authors do model-checking of the timed behavior of publish/subscribe applications using the Bogor model checker, which can be customized to different application domains [27]. Finally, in [28], the authors demonstrate the necessity of applying formal models to IoT protocols. In particular, they model the MQTT publish/subscribe protocol based on a timed message-passing process algebra. The analysis reveals that the protocol behaves correctly regarding the semantics of the QoS modes 1 and 2. However, the 3rd QoS mode is prone to error and at best ambiguous in certain aspects of its specification.

Our work relies on some of the formal modeling and analysis techniques of the above approaches. We further introduce a new timed model that abstracts the timing behavior of mobile IoT interactions across the existing heterogeneous IoT protocols. Our formal analysis based on timed automata and UPPAAL provides general conditions for successful interactions independently of the diverse underlying IoT protocols.

Alternatively or in addition to simulation based approaches, statistical model checking [29] may be applied in order to verify, for instance, probabilistic reachability properties. Nevertheless, simulation techniques are necessary as a starting point in order to elicit distributions needed as inputs to statistical model checkers. This is the case in [29], where authors perform simulations of a system in order to learn the application context. This creates a stochastic abstraction for the application, which is verified using statistical model checking. In the work done by Kim et al. [30], a formal specification is developed for each layer of a distributed system. To achieve the desired end-to-end timing/QoS properties, the formal specification is analyzed using statistical model checking and statistical quantitative analysis under various resource management policies. In our approach, we employ statistical quantitative analysis for evaluating performance properties of mobile IoT interactions by calculating, e.g., their mean values or probability distributions. Statistical model checking can be our next step for verifying or estimating the probabilities of such properties.

Overall in this paper, we unify the verification of the timing behaviors of DeX one-way, synchronous, asynchronous and streaming interactions, as well as the statistical performance analysis of such interactions. While our prior work focused on the timing and QoS analysis of heterogeneous service choreographies [4, 31], we model here the fine-grained effect of timing parameters on both coupled and decoupled mobile IoT systems. By leveraging our analysis of timing parameters, designers of IoT systems can accurately set constraints to ensure high success rates for interactions.

8 Conclusion and future work

Timing constraints have typically been used for time-sensitive systems to ensure properties such as deadlock freeness and time-bounded liveness. In this paper, we study IoT interactions using the DeX connector model which is enhanced to accurately model timing behavior through timed automata. Verification of conditions for successful DeX interactions is done in UPPAAL in conjunction with the timing guards specified. We demonstrate that accurate setting of *lifetime*, *timeout*, *time_on* and *serve_time* periods significantly affects the DeX interactions success rate. By providing a fine-grained analysis of the related timing parameters for designers of IoT applications, increased probability of successful interactions can be ensured. This is crucial for accurate runtime behavior, especially in the case of heterogeneous space-time coupled/decoupled DeX interactions with variable connectivity of IoT devices. Furthermore, we demonstrate that the response time vs. success rate trade off can be suitably configured. Finally, we confirm the sufficient accuracy of our results by comparing with experimental outcomes from the DeXM implementation framework.

The timed modeling and analysis of IoT interactions presented in this paper can open up directions for the design of robust and efficient IoT systems. In our future work, we intend to model the performance of realistic DeX interactions by considering processing and transmission delays, as well as the effect of DeXM mediators. We will leverage our experience in the modeling of publish/subsbribe protocols using queueing network models. Such models can provide us with both simulation and analytical models for estimating performance metrics such as response times and delivery success rates. Finally, these performance metrics can be leveraged for runtime system tuning and reconfiguration using heuristic and optimization techniques.

Abbreviations
IoT: Internet of things; APIs: Application programming interfaces; DeX: Data eXchange; DeXM: Data eXchange mediators; SOA: Service-oriented architecture; CS: Client/server; PS: Publish/subscribe; DS: Data streaming; TS: Tuple space; TIM: Transport information management; QoS: Quality of service; RPC: Remote procedure call; CTL: Computation tree logic; DPWS: Devices profile for web services; JMS: Java message service; QNs: Queueing networks; PPNs: Performance petri nets; MVA: Mean value analysis; HTA: Hierarchical timed automaton

Acknowledgements
We acknowledge colleagues of the INRIA MINES team for their suggestions.

Funding
This work has been partially supported by the European Union's Horizon 2020 project CHOReVOLUTION under grant agreement No. 644178.

Availability of data and materials
Our prototype and its modules are publicly available at: https://gitlab.inria.fr/dexms

Authors' contributions
All authors read and approved the final manuscript.

Declarations

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Télécom SudParis, Institut Polytechnique de Paris, France. [2]INRIA, Paris, France. [3]Ericsson Research, Bangalore, India.

Received: 19 April 2020 Accepted: 20 October 2021
Published online: 01 December 2021

References
1. Guinard D, Trifa V, Karnouskos S, Spiess P, Savio D. Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. IEEE Trans Serv Comput. 2010;3(3):223–35.
2. Fysarakis K, Askoxylakis I, Soultatos O, Papaefstathiou I, Manifavas C, Katos V. Which iot protocol? comparing standardized approaches over a common m2m application. In: IEEE Global Communications Conference (GLOBECOM). Washington: IEEE; 2016.
3. Bouloukakis G, Georgantas N, Ntumba P, Issarny V. Automated synthesis of mediators for middleware-layer protocol interoperability in the iot. Futur Gener Comput Syst. 2019;101:1271–94. https://doi.org/10.1016/j.future.2019.05.064.
4. Kattepur A, Georgantas N, Bouloukakis G, Issarny V. Analysis of timing constraints in heterogeneous middleware interactions. In: ICSOC. Goa: Springer; 2015.
5. Alur R, Dill DL. A theory of timed automata. Theor Comput Sci. 1994;126(2):183–235.
6. Behrmann G, David A, Larsen KG. A tutorial on Uppaal 4.0: Department of computer science, Aalborg university; 2006.
7. Schrank D, Eisele B, Lomax T. Tti's 2012 urban mobility report. 2012.
8. ITS. Intelligent Transportation Systems. http://www.flir.co.uk/traffic/content/?id=66601.
9. XD. Traffic. http://inrix.com/xd-traffic.
10. Yoon J, Noble B, Liu M. Surface street traffic estimation. In: ACM Mobisys. PR: ACM; 2007.
11. Mohan P, Padmanabhan VN, Ramjee R. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In: ACM SenSys. Raleigh: ACM; 2008.
12. Eugster P, Felber P, Guerraoui R, Kermarrec A. The many faces of publish/subscribe. ACM Comput Surv (CSUR). 2003;114–31.
13. Aldred L, van der Aalst W, Dumas M, ter Hofstede A. On the notion of coupling in communication middleware. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". Agia Napa: Springer; 2005.
14. Georgantas N, Bouloukakis G, Beauche S, Issarny V. Service-oriented Distributed Applications in the Future Internet: The Case for Interaction Paradigm Interoperability. In: ESOCC. Managa: Springer; 2013.
15. Bouloukakis G, Moscholios I, Georgantas N, Issarny V. Performance modeling of the middleware overlay infrastructure of mobile things. In: IEEE ICC. Paris: IEEE; 2017.
16. Bouloukakis G, Georgantas N, Kattepur A, Issarny V. Timeliness Evaluation of Intermittent Mobile Connectivity over Pub/Sub Systems. In: ACM/SPEC ICPE. L Aquila: ACM; 2017.
17. Bouloukakis G, Kattepur A, Georgantas N, Issarny V. Queueing network modeling patterns for reliable and unreliable publish/subscribe protocols. In: 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous). New York: ACM; 2018.
18. Clarke E, Emerson A, Sistla P. Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans Program Lang Syst (TOPLAS). 1986;8(2):244–63.
19. Lee K, Lee J, Yi Y, Rhee I, Chong S. Mobile data offloading: how much can wifi deliver?. In: Proceedings of the 6th International COnference. ACM; 2010.
20. Vernon M, Zahorjan J, Lazowska ED. A Comparison of Performance Petri Nets and Queueing Network Models: University of Wisconsin-Madison, Computer Sciences Department; 1986.

21. Kattepur A, Nambiar M. Performance modeling of multi-tiered web applications with varying service demands. In: IPDPS Workshops. Hyderabad: IEEE; 2015.
22. Kwiatkowska M, Norman G, Parker D. PRISM: Probabilistic symbolic model checker. In: MMMECCS. Aachen: Springer; 2001.
23. Nouri A, Bozga M, Legay A, Bensalem S. Performance evaluation of complex systems using the sbip framework. In: VECoS. Montreal: Springer; 2016.
24. Waszniowski L, Krakora J, Hanzalek Z. Case study on distributed and fault tolerant system modeling based on timed automata. J Syst Softw. 2009.
25. Zhou Y, Ge J, Zhang P, Wu W. Model based verification of dynamically evolvable service oriented systems. Sci China Inf Sci. 2016.
26. He F, Baresi L, Ghezzi C, Spoletini P. Formal analysis of publish-subscribe systems by probabilistic timed automata. In: FORTE. Tallinn: Springer; 2007.
27. Baresi L, Ghezzi C, Mottola L. On accurate automatic verification of publish-subscribe architectures. In: ICSE. Minneapolis: IEEE Computer Society; 2007.
28. Aziz B. A formal model and analysis of an iot protocol. Ad Hoc Netw. 2016;36:49–57.
29. Basu A, Bensalem S, Bozga M, Caillaud B, Delahaye B, Legay A. Statistical abstraction and model-checking of large heterogeneous systems. 2010.
30. Kim M, Stehr MO, Talcott C, Dutt N, Venkatasubramanian N. Combining formal verification with observed system execution behavior to tune system parameters. Quebec: Springer; 2007.
31. Kattepur A, Georgantas N, Issarny V. Qos analysis in heterogeneous choreography interactions. In: International Conference on Service-Oriented Computing. Springer; 2013.

Publisher's Note