View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 83 (2016) 195 – 202

Procedia Computer Science

The 7th International Conference on Ambient Systems, Networks and Technologies (ANT 2016)

A Transaction Model for Executions of Compositions of Internet of Things Services

K. Vidyasankar

Department of Computer Science,Memorial University, St. John's, Newfoundland, Canada A1B 3X5

Abstract

Internet of Things (IoT) is about making "things" smart in some functionality, and connecting and enabling them to perform complex tasks by themselves. The functionality can be encapsulated as services and the task executed by composing the services. Two noteworthy functionalities of IoT services are monitoring and actuation. Monitoring implies continuous executions, and actuation is by triggering. Continuous executions typically involve stream processing. Stream input data are accumulated into batches and each batch is subjected to a sequence of computations, structured as a dataflow graph. The composition may be processing several batches simultaneously. Additionally, some non-stream OLTP transactions may also be executing concurrently. Thus, several composite transactions may be executing concurrently. This is in contrast to a typical Web services composition, where just one composite transaction is executed on each invocation. Therefore, defining transactional properties for executions of IoT service compositions is much more complex than for those of conventional Web service compositions. In this paper, we propose a transaction model and a correctness criterion for executions of IoT service compositions. Our proposal defines relaxed atomicity and isolation properties for transactions in a flexible manner and can be adapted for a variety of IoT applications.
© 2016 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of the Conference Program Chairs

Keywords: Internet of Things services; transactions; atomicity; isolation; stream data processing;

1. Introduction

The concept of transactions has been extremely helpful to regulate as well as ensure the correctness of concurrent executions of operations in various applications. The transaction concept was introduced first in the context of (centralized) database systems, and then adopted in various advanced database and other applications, more recently in Web services [1], electronic contracts [2], transactional memory [3] and stream processing [4]. Transactions are characterized by ACID properties: Atomicity, Consistency, Isolation and Durability. While these properties are considered very strictly for database operations and memory operations [3], they are relaxed in other applications, depending on the se-

* K. Vidyasankar. Tel.: +1-709-864-4369; fax: +1-709-864-2009.
E-mail address: vidya@mun.ca

1877-0509 © 2016 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of the Conference Program Chairs
doi:10.1016/j.procs.2016.04.116

mantics and constraints of the application environments. The earliest and most universally applied relaxation is with atomicity and isolation, in the definition of *sagas*[5]:

- A transaction is said to be correct and to preserve consistency, if executed completely or not at all;
- A higher level transaction can be split into, and executed by, several lower level transactions;
- Then, isolation is relaxed from the entire high level transaction to the individual lower level transactions;
- For atomicity, all the lower level transactions must be executed successfully, or none at all;
- If some of them are execuited successfully, but others cannot be executed successfully, then the earlier ones need to be compensated, to achieve overall null execution; and
- The compensation can only be logical and should take into account that other transactions might have observed and used the results of the successfully executed low level transactions.

Atomicity and isolation have been relaxed further in the various contexts mentioned above and in general nested transactions. In this paper, we study the relaxations that are applicable to compositions of Internet of Things Services.

Internet of Things (IoT) is about things connected through internet. A "thing" is any object of interest with some communication capability. Following the Service Oriented Architecture (SOA), "service" encapsulation is given to the functionality of things. These services can be employed as any other services. Basic services are combined to perform complex tasks or to build composite services. Monitoring and actuation are two essential functionalities of IoT services. Monitoring implies continuous executions, and actuation is by triggering. Continuous executions typically involve stream processing. Stream input data are subjected to a sequence of computations, structured as a dataflow graph. The computation is push-based: as soon as batches of input arrive, the computation is triggered. In addition, the stream input keeps arriving continuously and so the composition may be executing several batches concurrently. Additionally, some non-stream transactions (denoted OLTP transactions in this paper) may also be executing concurrently. In contrast, in a typical Web services composition, the execution is pull-based and we normally consider one-time execution (of an OLTP transaction) in each invocation of the composition.

In this paper, we propose a transaction model and a correctness criterion for executions of IoT services compositions. Our proposal defines relaxed atomicity and isolation properties in a flexible manner and can be adapted for a variety of IoT applications. We introduce executions of IoT services compositions in Section 2. We start with core definitions of compositions and transactions in this section. Then, we discuss the various relaxations in Section 3, and give our transaction model and a composition model in that section. We discuss related work in Section 4 and conclude in Section 5.

2. Executions

A *composition* C is $(\mathcal{P},\prec_p)$, where $\mathcal{P}$ is a set of *transaction programs* $\{P_1, P_2, \ldots, P_n\}$, simply called *programs*. and $\prec_p$ is a partial order among them. We call the (acyclic) graph representing the partial order the *composition graph* $\mathcal{GC}$. Each execution of a program yields a *transaction*. A transaction may have some stream and/or non-stream inputs, and may produce some stream and/or non-stream outputs. Stream data are sequences of tuples. They may come from outside the composition; these are from *base* streams. Others may be generated by programs in the composition; these are of *derived* streams. Transactions process stream inputs in *atomic batches* (also referred to, simply, as *batches*), the batch size being determined by applications. The stream outputs produced by transactions are also in batches, and they will constitute atomic batches for inputs to other transactions, if any.

In an execution of a composition, some of its programs will be executed, resulting in a set of transactions with a partial order $\prec_t$. We call this a *composite transaction*, denoted as $\mathcal{T} = (\{T_1, T_2, \ldots, T_m\},\prec_t)$. We denote $\{T_1, T_2, \ldots, T_m\}$ as *set*$(\mathcal{T})$. The graph representing $\prec_t$ is called *transaction graph* $\mathcal{GT}$. The transaction graphs are acyclic. We note that each T_i is an execution of some program P_j. It is possible that $\mathcal{T}$ has more than one execution of some P_j (like in Meehan et al[6]). The partial order $\prec_t$ is compatible with $\prec_p$, that is, if T_i is an execution of P_j, T_k is an execution of P_l and $P_j \prec_p P_l$, then $T_i \prec_t T_k$.

Stream processing is usually push-based. As soon as the stream input tuples add up to a batch, the execution of the program for which they are input will start. OLTP transactions may be executed in pull-based fashion, like in Web services composition dealing with non-stream data. Both stream processing and OLTP transactions may trig-

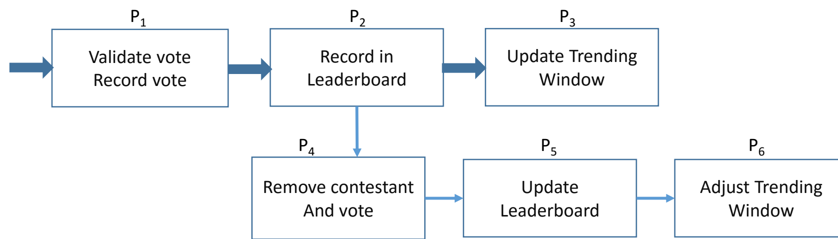

Fig. 1. Leaderboard Example

ger execution of other transactions, that is, execution of other programs. The partial order in the composition graph includes (i) data flow orders of the streams, (ii) the order defined between stream processing transactions and OLTP transactions, and among OLTP transactions, (referred to as *control* order in this paper) and (iii) the triggering relationships. Unless explicitly distinguished, we refer to all of these collectively as triggering relationships. Composite transactions inherit the ordering relationships from the composition. Thus, in a composite transaction, a transaction T_i may precede another transaction T_j due to any of the above three partial orders.

Executions of a composition may be triggered either by the arrival of an atomic batch of stream input or by a OLTP-type invocation in the traditional sense of composition execution. We call the former as *stream composite transactions* and the latter as *OLTP composite transactions*. In either case, several transactions will be executed, respecting the partial order.

We denote the composite transaction executed with batch b as $\mathcal{T}(b)$. Stream input batches arrive in sequence, for example, as $b_1, b_2, \ldots$. The batch order is denoted $\prec_b$. The batch b_2 and some more batches may arrive before all the transactions in $\mathcal{T}(b_1)$ are completely executed. Thus many stream transactions may be executing concurrently. Some OLTP composite transactions may also be executing concurrently. This is in contrast to typical Web services compositions, where we will be concerned with only one composite transaction at a time.

We illustrate the above definitions with an example composition.

Example 1: *Leaderboard Maintenance*[6]. The general description of the problem is from that paper. We simplify and modify the problem slightly.

"Consider a TV game-show in which viewers vote for their favorite candidate. Leaderboards are periodically updated with the number of votes each candidate has received. Each viewer may cast a single vote via text message. Suppose the candidate with the fewest votes will be removed from the running every 20,000 votes, as it has become clear that s/he is the least popular. When this candidate is removed, votes submitted for him or her will be deleted, effectively returning the votes to the people who cast them. Those votes may then be re-submitted for any of the remaining candidates. This continues until a single winner is declared."

We keep one leaderboard, listing all the current contestants and the votes obtained by them. We also consider groups of every 100 votes, and find out the top 3 candidates in each group. The results of the most recent group are displayed in a trending window. (The results will remain until the next group results are computed.) With each incoming vote, the vote is validated and these boards are updated appropriately. We will also allow a candidate to withdraw from the contest voluntarily (for whatever reasons). In that case, we simply remove that candidate from leaderboard, and also from the trending window, if necessary.

Figure 1 depicts the composition, namely, the programs and the partial order among them (which, as mentioned above, includes stream data flow order, control order and triggering order). In this example, each vote constitutes an atomic batch. Processing details are as follows:

- The input stream is made up of the votes.
- Processing is done by a workflow consisting of the following programs:
  - P_1 validates (checking that both the user and the contestant are legitimate, the user is eligible to vote and the contestant has not been removed from the contest) and records votes;

  - P_2 enters the vote in the leaderboard; and
  - P_3 updates trending window, after every 100 votes.
- P_2 triggers P_4, once every 20,000 votes, for removing lowest contestant and his/her votes; the contestant's id is passed as a parameter.
- The actual removal is done by P_4. ("Returning votes" to the viewers is not modelled.)
- The resulting updates in the leaderboard and trending window are performed by P_5 and P_6, respectively.
- Two stored tables `VOTES` and `CONTESTANTS` are used in the execution of the programs.

. For an atomic batch b_j, the execution of P_i is denoted T_{ij}. Then, for a batch b_j, the transactions in $\mathcal{T}(b_j)$ may contain $\{T_{1j}, T_{2j}\}$, $\{T_{1j}, T_{2j}, T_{3j}\}$, or $\{T_{1j}, T_{2j}, T_{3j}, T_{4j}, T_{5j}, T_{6j}\}$. We note that T_{1j} and T_{2j} will be performed for all the tuples, T_{3j} will be performed after every 100 tuples, and T_{4j} will be triggered after every 20,000 votes. We assume that when T_{4j} is triggered, T_{3j} will also be executed. T_{6j} will perform update of the trending window, if needed.

An OLTP composite transaction that is initiated when a contestant voluntarily quits the contest will contain $\{T_{4j}, T_{5j}, T_{6j}\}$.

3. Transaction Model

As mentioned earlier, several composite transactions may be executing concurrently in a composition. General (strict) requirements for correct concurrent executions can be stated as follows. Here, let $\mathcal{T}$ be $(\{T_1, T_2, \ldots, T_m\}, \prec_t)$.

1. *Unit of atomicity*: The atomicity requirement is that each composite transaction is (effectively) executed either completely or not at all. That is, the entire $\mathcal{T}$ is an *atomic unit* for each $\mathcal{T}$.
2. *Serializability*: The execution is equivalent to a serial execution of the composite transactions.
3. *Transaction order*: The effective execution order of the transactions of $\mathcal{T}$ should obey the partial order $\prec_t$. That is, for any i, j, if $T_i \prec_t T_j$, then T_i should precede T_j in the serial execution.
4. *Batch order*: The serial execution should reflect the batch order $\prec_b$. That is, for i, j, (all the transactions in) $\mathcal{T}(b_i)$ should precede (the transactions of) $\mathcal{T}(b_j)$ in the serial execution. OLTP composite transactions may occur in any order in the serial execution, relative to the stream composite transactions.
5. *Completion*: For each $\mathcal{T}$, all T_i's, for $1 \leq i \leq m$, must be executed. And, if **T** is the set of composite transactions under consideration, all of them must be executed.
6. *Monotonic execution*: At any time, the executed schedule must be such that, for any composite transaction $\mathcal{T}$, its projection on the completed transactions of $\mathcal{T}$ should be a prefix[1] of the transaction graph $\mathcal{GT}(\mathcal{T})$.

In most applications, the transactions will be executed in distributed fashion. Satisfying the above requirements would be impossible or, at the very least, will yield very poor performance. The semantics of the application may be such that many of those requirements could be relaxed. In the following, we look into the ways in which the requirements can be relaxed. We illustrate with our leaderboard and other examples.

(1) *Unit of atomicity*: In many applications, subsets of the transactions in $\mathcal{T}$ can be considered as units of atomicity. In the leaderboard example, recording the vote and entering it in the leaderboard might be considered as the most critical operations. Then, for the executions listed above, the set of atomic units could be $\{\{T_1, T_2\}, \{T_3\}, \{T_4, T_5, T_6\}\}$. In addition, when a contestant drops out, if a delay in leaderboard and trending window updates can be tolerated, then, instead of $\{T_4, T_5, T_6\}$, we could simply have $\{T_4\}$, $\{T_5\}$ and $\{T_6\}$ as atomic units.

We partition $\mathcal{T}$ into a set of atomic units $\mathcal{T}^a$. The graphs consisting of the transactions and the partial order among them of each of the atomic units would form a partition of $\mathcal{GT}(\mathcal{T})$.

(2) *Serializability*: Now, serializability can be with respect to the atomic units. That is, an equivalent serial execution need only have all the transactions in each atomic unit occur consecutively. In the leaderboard example, this

[1] A subgraph H of an acyclic graph G is a prefix of G if all the edges from H to the rest of the graph are outdirected.

implies, for instance, that in the composite transaction where the trending window is updated in T_3, the update might occur after T_1 and T_2 have been executed for a few more votes.

(3) *Transaction order*: If some inconsistency can be tolerated, the transaction order need not be followed for some transactions. We allow this flexibility in the level of atomic units, rather than individual transactions. For example, with $\{T_5\}$ and $\{T_6\}$ as atomic units, the transaction order between them need not be followed; that is, T_6 could be executed before T_5. We denote the resulting relaxed partial order as $\prec_a$. We note that this order will be compatible to $\prec_t$. That is, if $A_i \prec_a A_j$, for atomic units A_i and A_j, then for some transactions T in A_i and T' in A_j, $T \prec_t T'$.

(4) *Batch order*: In the leaderboard example, the arrival order of the tuples (votes from the viewers) in the base stream input is arbitrary and may not be important with respect to execution of the composition. Then, the batch order is not important, and hence can be ignored. We denote the batch order that must be kept as $\prec_{b'}$. This is for all the stream composite transactions. It is also possible to further relax the batch order requirement for some transactions or some atomic units, and not for others. An example is requiring batch order for $\{T_1, T_2\}$, and not for $\{T_3\}$, in the leaderboard example. In addition, if a delay in recording votes in the leaderboard is acceptable, the atomic units would just be $\{T_1\}$ and $\{T_2\}$. This, along with not requiring batch order, would allow two votes being validated in one order but entered in the leaderboard in a different order. The batch order could also be relaxed just for some batches.

We define the relaxed batch order requirement for atomic units and denote it as $\prec_{B'}$.

(5) *Completion*: (a) The first requirement is that, for each composite transaction $\mathcal{T}$, all its constituent transactions should be executed. As mentioned earlier, in most applications, the programs will be executed at different sites in a distributed fashion. In particular, a triggering transaction and its triggered transactions may be executed in different places. It may not be known when the triggered transactions will be executed or even whether they will be executed at all. We relax the completion requirement to execution of a prefix of the transaction graph and denote this as $\chi(\mathcal{T})$. Note that this is defined for each individual composite transaction. Thus, the sets could be different for different $\mathcal{T}$'s.

Triggering introduces another situation. We illustrate with the following example.

Example 2: *Assisted Living* [7]. Here, ALS refers to "Assisted Living System" proposed in that paper.

"Emily, 70 years old, has diabetes and she needs to take care of blood sugar level. The doctor prescribed her to use ALS biomedical sensors to monitor remotely her sugar level and vital signs. One day, while she is having a small headache and feeling dizzy, an abnormal data (violating the predefined sugar level threshold) is reported from her wearable unit; the ALS detects this situation and alerts the physician at the hospital/clinic and also sends a short text message to Emily's son about her abnormal health condition. The remote physician triggers the analysis of sugar, insulin, O_2 saturation in her blood; gets the results and sends Emily an SMS with the medical prescription and other dietary recommendations. A few minutes later her son visits Emily at home; one hour later, the physician checks Emily's vital signs and everything is back to the normal values."

We note that (i) several actions (transactions) are triggered, some of them triggering further actions and (ii) the triggered actions are executed at different sites. The triggerings (alerting the physician who, in turn, triggers the blood analysis) are critical. To ensure that they will be acted upon quickly, the triggerings could be repeated. (For example, the physician may not notice the alert quickly, and so it is better to keep alerting until the physician notices.) This could be while processing the subsequent atomic batches (of the sensor values in our example).

We suggest a way of accommodating this in our transaction model. With each transaction, along with its output (and input), we state explicitly the transactions it triggers, as *triggered transaction set*, or simply *tts*. Note that each *tts* is a subset of *set*($\mathcal{T}$). In our leaderboard example, this set would be $\{T_2\}$ for T_1, $\{T_3, T_4\}$ for T_2 (in the executions where both these are triggered), and so on. Thus a transaction T_i will trigger T_j whenever the conditions for triggering are satisfied in T_i. For "at least once" semantics, it is enough to execute T_j once, even though it is triggered multiple times. (We note that in some other applications the semantics may require that a triggered transaction is executed as many times as it is triggered.)

(b) The second completion requirement is that all composite transactions in $\mathbf{T}$, the set under consideration, must be executed. It follows that the stream composite transactions for all the atomic batches must be executed. It may not be possible to satisfy this requirement in many applications. If so, then consistency and accuracy of the underlying computation may suffer. The results will then be approximate. This may be tolerable in some applications. An example is losing sensor data intermittantly while tracking a weather system. Dropping (shedding) some atomic

batches may even be unavoidable, for example, when the stream input arrival rate is too high for the available resources to process. Quick, though approximate, answers may be preferred compared to delayed (probably obsolete) accurate answers. To improve the quality of the results, the number of batches dropped should be minimized. Even the actual batches that can be dropped could be qualified.

We illustrate with another example.

Example 3: *Emergency Management*[8].

"Consider an emergency management application that provides logistics information to police and fire companies as well as to the general public. Under normal conditions, the system can easily keep up with the load and display information to everyone who asks. However, when disaster strikes, the load can increase by orders of magnitude and exceed the capacity of the system. Without load shedding, the requests would pile up, and nobody would get timely responses. Instead, it is preferable to shed some of the load by only providing complete and accurate replies to requests from police or fire companies and degrading accuracy for everyone else."

Here, some input tuples are not used. The shedding is selective, based on who requests and/or for which updates.

Shedding of a batch b can also be specified in terms of $\chi(\mathcal{T}(b)$, the prefix of the transaction graph of $\mathcal{T}(b)$ that needs to be executed. If the prefix is null, then the dropped batch is from base stream, otherwise it is from a derived stream (which is an output of the prefix).

(6) *Monotonic execution*: The stated property, that, at any time, the parts of the composite transactions that have been executed successfully should be prefixes of their respective transaction graphs, follows from the requirement that the transaction order should be followed in the execution.

In addition, we observe the following. In the middle of execution of a composite transaction $\mathcal{T}$, we may find that some transaction in $\mathcal{T}$ cannot be executed successfully. Then, for atomicity, that is, "all or nothing" property, the transactions that have been executed so far must be rolled back, that is, compensated. Monotonic execution implies that the compensation must be done in the reverse order.

The above applies also when a complete $\mathcal{T}$ has to be rolled back, for whatever reasons. For example, in the leaderboard example, if it was realized sometime later that a vote, constituting a batch b, has to be disqualified, and in the earlier execution of $\mathcal{T}(b)$, T_1 and T_2 were executed, then the compensation order should be T_2 first and then T_1. If this is done the other way, then inconsistency (in the form of an invalid vote counted in the leaderboard) occurs, though only temporarily. We specify a compensation order for the atomic units as $\prec_{\bar{a}}$. Usually, this will be the inverse of $\prec_a$.

Note also that, since several composite transactions may be executed concurrently, roll back of one $\mathcal{T}$ can cause roll back of some other, conflicting, transactions. That is, cascading roll back may be warranted. The compensation order needs to be followed for each roll back.

Accommodating all the above features, we define a composite transaction as follows.

Definition: A *composite transaction* $\mathcal{T}$ of a composition C is $(\{T_1, T_2, \ldots, T_m\}, \prec_t)$, where each T_i is a partially ordered set of operations and has sets of inputs, outputs and triggered transactions, that is, *in*(T_i), *out*(T_i) and *tts*(T_i). A *complete specification* of $\mathcal{T}$ is $(\mathcal{T}^a, \prec_a, \prec_{B'}, \chi(\mathcal{T}), \prec_{\bar{a}})$.

We now consider similar model for a composition.

In Emily's health monitoring example (Example 2), the biomedical sensors will keep sending values periodically and hopefully they will be recorded somewhere for the physician's perusal. When abnormal conditions are detected, the physician is alerted. After that, the physician might like to receive the sensor values as they come, to monitor Emily's conditions in real time. Later, when s/he is satisfied that her conditions have become normal, s/he may like to stop receiving the sensor values.

The above example illustrates that the stream flow in the composition may change, and hence the programs executed for an atomic batch may change. We can model this simply by extending the triggered transaction set idea to programs. With each program in the composition, we associate a *triggerable program set*, called *tps*. This set will contain the programs that can be triggered (by any of the three means: due to stream flow; control order or transaction triggers) by that program. Then, the *tts* of a transaction resulting from the execution of a program in the composition will be a subset of *tps* of that program. This strategy allows changing the stream flow any number of times.

So far, we considered that the stream input comes from sources outside of the composition. It is quite possible that a stream is generated inside a composition. For instance, in the leaderboard example, for each area code, the number of people voted from the same area code can be calculated by another program P_7, using the information in the VOTES table and sent to some place every 10 minutes. This can be accommodated by reminding itself, by way of including P_7 itself, in the *tps* of P_7. Then, the transaction, say T' (an execution of P_7), that generates the stream will include T' in its *tts*. This will continue until the stream generation is stopped. Note that we allow for a triggered transaction to be executed some time after it is triggered. The time of the execution may very well depend on some other conditions like (i) after certain duration (for example, every 10 minutes) and (ii) other triggerings. Therefore, including a program in its own *tps* will not create infinite cycle. The point (ii) above facilitates the case where a transaction will be executed only when it is triggered by several transactions. In Emily's healthcare example, on noticing that the vital signs have not improved *and* the lab analysis result is not good, the physician may initiate further action such as admitting Emily in the hospital.

Our proposal is as follows.

Definition: A *composition* C is ($\mathcal{P}$, $\prec_p$), where each P_i in $\mathcal{P}$ has sets of inputs, outputs and triggerable programs, that is, *in*(P_i), *out*(P_i) and *tps*(P_i).

An execution of a composition consists of executions of several composite transactions say, $\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_m\}$, each consisting of a set of transactions that are executions of the programs in the composition. Denoting this set as **T**, We depict the execution in a graph $\mathcal{GE}$ (**T**)called the *execution graph* of **T**. It will contain the transaction graphs of all the composite transactions in **T**, and in addition, edges corresponding to $\prec_b$. Thus, the nodes of $\mathcal{GE}$ (**T**) will be the transactions resulting from the executions of programs in the composition, and the edges corresponding to $\prec_t$ and $\prec_b$.

Definition: An execution of a composition is correct if, at any time, the committed projection corresponds to a prefix of its execution graph.

4. Related Work

Several papers discussing compositions in different environments (Transactional processes [9], Web services [1], Electronic contracts [2] and Transactional memory [3]) have been mentioned in the Introduction section. All these compositions are for one-time execution per invocation. Other works include the following.

- A streaming service communication model, an infrastructure supporting composition of streaming services and execution using that model, and a platform, called ComSS [10] which acts as a middleware for running composite streams' processing services;
- Discussion of transactional stream processing [4] and the proposal of a unified transaction model, called UTM, that treats events also as transactions. Atomicity and isolation properties for transactions in this model are discussed in detail in the paper.
- Discussion of events and triggers in the context of Complex Event Processing over Event Streams [11]. They also define *stream* ACID properties for transactions: *s*-Atomicity, *s*-consistency, *s*-Isolation and *s*-Durability. The *s*-Atomicity notion requires "all operations stimulated by a single input event should occur in their entirety". That is, a triggering transaction as well as all transactions triggered by them form a single unit of atomicity. We feel this notion is inappropriate in the IoT environment, since the triggering and the triggered transactions may be executed "far away" from each other, in different sites that are autonomous. Furthermore, triggering can go on to multiple levels. Combining triggering and all the triggered transactions in all levels is an impossible task. This is the reason we separated triggering and triggered transactions into different atomic units.
- Transactional execution of stream composition in S-Store [6]. In that paper, the unit of atomicity is the entire composite transaction.
- Role of transactions for self-healing IoT applications [12]. In that paper, an application is modelled as a colored Petri Net. Triggering amounts to depositing a token in the output place. Our *tts* and *tps* formulations abstract this idea.
- Other papers discussing stream transactions and compositions [13,14,15,16,17].

5. Conclusion

The basic premise of the Service Oriented Architecture is to treat all functionalities as services, and compose and execute them according to user or application requirements. Treating each service execution as a transaction and requiring atomic execution of those transactions have been found very helpful in this process. Then, executions of service compositions yield composite transactions.

Compositions are typically executed in a distributed fashion in several 'sites'. The autonomy of the sites makes achieving atomicity, that is either complete successful execution of the entire composite transaction or the null execution, very difficult. Therefore, the atomicity and isolation properties need to be relaxed. We have considered relaxations that are appropriate to IoT services compositions in this paper. The relaxations are with respect to (i) restricting atomicity to parts of the transaction, (ii) preserving composition and batch order, and (iii) executing transactions only partially. Our proposal takes into account the special requirements arising out of stream processing and triggering, and the concurrent executions of several stream and OLTP composite transactions in IoT service compositions. The notion of correctness of concurrent executions stated in this paper is sufficiently general and can be tailored to specific requirements of a variety of IoT application scenarios.

Acknowledgment

This research is supported in part by the Natural Sciences and Engineering Research Council of Canada Discovery Grant 3182.

References

1. K. Vidyasankar, G. Vossen, Multi-level modeling of web service compositions with transactional properties, Database Management 22 (2) (2011) 1–31.
2. K. Vidyasankar, P. R. Krishna, K. Karlapalem, A multi-level model for activity commitments in e-contracts, in: Proceeding OTM Conferences, 2007, pp. 300–317.
3. S. Peri, K. Vidyasankar, Correctness of concurrent executions of closed nested transactions in transactional memory systems, Theoretical Computer Science 496 (2013) 125–153.
4. I. Botan, P. M. Fischer, D. Kossmann, N. Tatbul, Transactional stream processing, in: Proceedings EDBT, ACM Press, 2012.
5. H. Garcia-Molina, K. Salem, Sagas, in: Proceedings of theACM SIGMOD International Conference on Management of Data, ACM Press, 1987, pp. 249–259.
6. J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, H. Wang, S-store: Streaming meets transaction processing, Proc. VLDB Endow. 8 (13) (2015) 2134–2145.
7. K. Dar, A. Taherkordi, R. Vitenberg, R. Rouvoy, F. Eliassen, Adaptable service composition for very-large-scale internet of things systems, in: Proceedings of the Workshop on Posters and Demos Track, PDT '11, ACM, New York, NY, USA, 2011, pp. 11:1–11:2.
8. M. Hirzel, R. Soulé, S. Schneider, B. Gedik, R. Grimm, A catalog of stream processing optimizations, ACM Comput. Surv. 46 (4) (2014) 46:1–46:34. doi:10.1145/2528412.
   URL `http://doi.acm.org/10.1145/2528412`
9. H. Schuldt, G. Alonso, C. Beeri, H. Schek, Atomicity and isolation for transactional processes, ACM Transactions on Database Systems 27 (2002) 63–116.
10. P. Świątek, ComSS - platform for composition and execution of streams processing services, in: ACIIDS 2015, Part II, Lecture Notes in Computer Science 9012, Springer International Publishing Switzerland, 2015, pp. 494–505.
11. D. Wang, E. A. Rundensteiner, R. T. E. III, Active complex event processing over event streams, in: Proceedings of the VLDB Endowment, ACM Press, 2011, pp. 634–645.
12. R. Angarita, Responsible objects: Towards self-healing internet of things applications, in: IEEE International Conference on Autonomic Computing (ICAC), Grenoble, 2015, pp. 307–312. doi:10.1109/ICAC.2015.60.
13. L. Golab, M. Özsu, Issues in data stream management, ACM SIGMOD Record 32 (2) (2003) 5–14.
14. A. I. Luigi Atzori, G. Morabito, The internet of things: A survey, Computer Networks 54 (15) (2010) 2787–2805.
15. N. Conway, Transactions and data stream processing, in: Online Publication, http://neilconway.org/docs/stream_txn.pdf, 2008, pp. 1–28.
16. L. Gürgen, C. Roncancio, S. Labbé, V. Olive, Transactional issues in sensor data management, in: Proceedings of the 3rd International Workshop on Data Management for Sensor Networks (DMSN'06), Seoul, South Korea, 2006, pp. 27–32.
17. M. Oyamada, H. Kawashima, H. Kitagawa, Continuous query processing with concurrency control: Reading updatable resources consistently, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, ACM, New York, NY, USA, 2013, pp. 788–794. doi:10.1145/2480362.2480514.
   URL `http://doi.acm.org/10.1145/2480362.2480514`