2011 10th IEEE/ACIS International Conference on Computer and Information Science

Atomicity Maintenance in EPCreport of ALE

Jing Sun*, Huiqun Zhao*, Wenwen Wang*, Gongzhu Hu†

*Department of Computer Science, North China University of Technology, Beijing, China
†Department of Computer Science, Central Michigan University, Mount Pleasant, Michigan 48859, USA

***Abstract*—The Application Layer Event (ALE) is an interface of Electronic Product Code (EPC) network framework launched by EPCglobal, through which clients may obtain filtered consolidated EPC. The ALE provides a poll mode for creating a synchronous report. Under the EPC network framework the poll mode has to maintain atomicity for event cycle from different read cycles. It is a big challenge for decentralized EPC network to long-term block channel for synchronous report. In this paper we propose two approaches to address this challenge. The first is to use a buffering storage to relax the ALE engine from the bursting of EPCs cycles. After all read cycles belonging to the same event cycle completed, the EPC engine sends an EPCreport to its consumer in an asynchronous mode. The second strategy is to apply multi-level granularity to divide a coarse event cycle into fine-grained cycles. With this strategy, all sub-EPCreports received by the consumers will be reorganized into a single EPCreport. We propose two algorithms and tested them using a simulation tool to show the performance efficiency.**

Keywords: EPC Network, EPCreport, Atomicity.

I. Introduction

As a standards organization, EPCglobal published a framework for developing applications of Electronic Product Code (EPC) network in 2005 [9]. In order to leave flexibility to it user, it only gives a series of interfaces for implementation. The Application Layer Event (ALE) [8] is one of the interfaces through which clients may obtain filtered consolidated EPC data from a variety of Radio Frequency Identification (RFID) readers. The objective of ALE is to reduce the volume of EPC data that comes directly from RFID readers into applications. According to [8], "the processing done at this layer typically involves: (1) receiving EPCs from one or more RFID readers; (2) accumulating data over intervals of time, filtering to eliminate duplicate EPCs and EPCs that are not of interest, and counting and grouping EPCs to reduce the volume of data; and (3) reporting in various forms," such as XML and database. The ALE model is illustrated in Fig. 1.

In Fig. 1, there are three read cycles identified as $read\ cycle_{1,2,3}$. A *read cycle* is the smallest unit of interaction with a reader. The result of a read cycle is a set of EPCs. An *event cycle* is one or more read cycles, from one or more readers that are to be treated as a unit from the client perspective. Four different types of EPC events may appear in an event cycle according to the EPCglobal standards: `TransctionEvent`, `QuantityEvent`, `ObjectEvent` and `AggregationEvent`. The `TransactionEvent` represents an event in which one or more entities denoted by EPCs become associated or disassociated with one or more identified business transactions. There are only two event cycles presented in Fig. 1. Finally, the ALE reports to its client what kind of EPC data has been captured according to the capture specification.

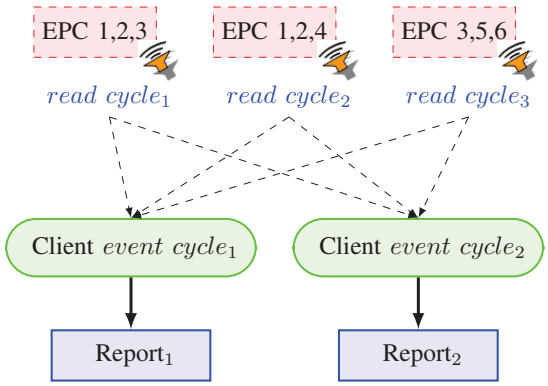


Fig. 1: A model of ALE

As a technique standard, EPCglobal provides an API to facilitate development of ALE applications. For example, `poll(specName:string):ECReports` is one of the API methods that we shall discuss in this paper with an implementation. The API specifies the atomicity property constraint by a case study while leaving the flexibility of implementation up to the developers.

An important issue the developer has to deal with is the atomicity property of event cycle. That is, an event cycle is treated as a single atomic unit and proper handling failure of a read cycle (as part of an event cycle) is critical. Quite different from transaction process in databases, if a read cycle failed the EPC data will be lost according to EPCglobal standards. So there is no way to undo a completed operation and reverting the system back to its previous state.

To address this problem, a fault tolerant strategy is proposed in this paper. We give a new architecture for ALE with respect to fault tolerance and design several algorithms for developing `poll(specName:string):ECReports`. We also evaluate the performance of the proposed algorithms with simulation examples.

II. Architecture Design of ALE Implementation

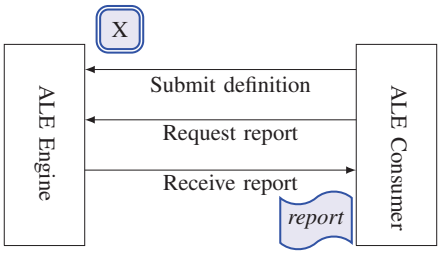
In this section we use a series of charts to demonstrate the design of a software component that implements ALE. We focus on the atomicity maintenance during the EPC data collection and filtering.

978-0-7695-4401-4/11 $26.00 © 2011 IEEE
DOI 10.1109/ICIS.2011.42

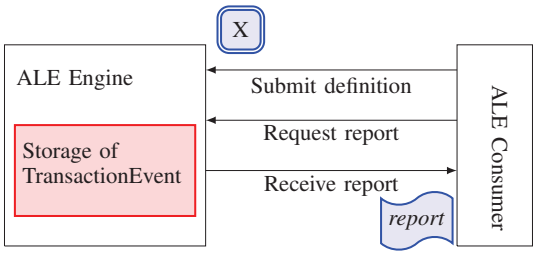
IEEE computer society

A. Atomicity Maintenance by Redundancy

Fig. 2a shows a poll mode of on-demand synchronous report from a standing request. An ALE consumer submits a report definition as an XML document or some other type of description. Once a request is triggered the ALE Engine will immediately create a report for its consumer. Since a coarser event cycle may be involved in the report definition, the ALE consumer has to be blocked to wait for a response during the duration of the event cycle.



(a) synchronous report



(b) asynchronous report

Fig. 2: Pool modes

Fig. 2b is a new mode for poll method. It includes some of the same systematic components as in the synchronous poll mode, and an additional Storage component that is used for `TransacitonEvent`.

In this new method, all sets of EPCs that belong to one `TransactionEvent` are stored in a database to save guard the EPCreports from being lost. The ALE consumer's request for a single report has to wait until all the EPCs have been accumulated. For this reason the new mode employs an asynchronous way to serve the consumer.

We first give formal specifications of the poll mode. For simplicity, LOTOS (Language of Temporal Ordering Specification) [19] is used to describe the two types of poll mode: asynchronous and asynchronous. We will then describe the two algorithms that we developed for implementing the poll methods based on the formal specifications.

The synchronous poll mode is given in Specification 1, and asynchronous pool mode is described in Specification 2.

B. Atomicity Maintenance by Fine-grained Event Cycle

Fig. 3 is an improved model of ALE for atomicity maintenance. In this model we divide a coarser event cycle into a set of fine-grained event cycles, each of which has its own EPCreport using the original synchronous pool mode. Once ALE's consumer receives all EPCreports from fine-grained event cycles that belong to same coarse event cycle as defined in the Report Definition, it reorganizes them according to the report definition. Similar to Fig. 2b, the ALE consumer should add a storage component where the received EPCreports are stored.

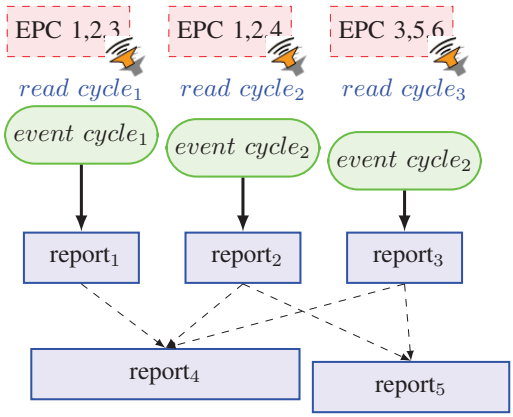


Fig. 3: Architecture of ALE for fine-grained event

The LOTOS description is given in Specification 3, the process *Poll_Sync_2*.

III. Atomicity Maintenance Algorithms

In this section, we first give a formal description of EPCreport for poll method output, and then use the description in the poll algorithms. In the following, $S = \{s_1, s_2, \dots\}$ is a set of EPC codes.

Definition 1. A grouping operator is a function G that maps an EPC code to a group code g, denoted $S \downarrow g = \{s \in S \,|\, G(s) = g\}$, the subset of EPCs $s_1, S_2 \cdots \in S$ that belong to group g.

For example, a grouping operator might map an EPC code into a Global Trade Item Number (GTIN) group.

Definition 2. A filtering operator is a function F that filters a set of EPC codes into a group code f, denoted $S \downarrow f = \{s \in S \,|\, F(s) = f\}$, the subset of EPCs $S_1, S_2 \cdots \in S$ that belong to group f.

For example, a filtering operator might filter EPC code coming from read cycles based on transaction event definition, i.e. for an event cycle definition.

Definition 3. A group membership report for one or more specified grouping operators G_i, or Filter operators F_i which may include, and possibly be limited to, the default group. Denoted $\{(g, E(R)\downarrow g \,|\, E(R)\downarrow g \neq \varnothing)\}$, E is either G_i or F_i operators.

Definition 4. A group cardinality report for one or more specified grouping operators G_i, or Filter operators F_i which may include, and possibly be limited to, the default group. Denoted: $\{(g \,|\, E(R) \downarrow g \,|) \,|\, E(R) \downarrow g \neq \varnothing)\}$, E is either the G_i or the F_i operator.

Specification 1 Synchronous pool mode

```
specification Poll_Sync [request, report, Channel_1, Channel_2, Channel_3] : noexit
    library
        BOOLEAN, NATURAL
    endlib
behaviour
    ALE_Consumer [request, report]
        | [request, report] |
    ALE_Engine [request, report, Channel_1, Channel_2, Channel_3]
        | [Channel_1, Channel_2, Channel_3] |
        (
            Reader_1 [Channel_1]
        |||
            Reader_2 [Channel_2]
        |||
            Reader_3 [Channel_3]
        )
process ALE_Engine [request, report, Channel_1, Channel_2, Channel_3] : noexit :=
    request ? read cycle_1 : Nat ? read cycle_2 : Nat ? read cycle_3 : Nat;
    (
        ([read cycle_1 == 1] && [read cycle_2 == 2] && [read cycle_3 == 3]) -> /* Case for EPCreport1.
            Channel_1 ? g : Nat; /* get EPCs from Channel
            Channel_2 ? h : Nat;
            Channel_3 ? j : Nat;
            report !g !h !j;   /* g.h.j represent EPCreports. Case for EPCreport2.
    []
        ([read cycle_1 == 2] && [read cycle_2 == 3] && [read cycle_3 == 0]) ->
            Channel_2 ? h : Nat;
            Channel_3 ? j : Nat;
            report !h !j;
    )
```

Specification 2 Asynchronous pool mode

```
specification Poll_Async [request, report, Channel_1, Channel_2, Channel_3] : noexit
    ..... The main structure is the same as Pool_Sync.
process ALE_Engine [request, report, Channel_1, Channel_2, Channel_3] : noexit :=
    request ? read cycle_1 : Nat ? read cycle_2 : Nat ? read cycle_3 : Nat;
        Client_Event [request, report, Channel_1, Channel_2, Channel_3]
        /* Call sub_process for completing Asynchronous poll mode.
    .....
process Client_Event [request, report, Channel_1, Channel_2, Channel_3] : noexit :=
    Channel_1 ? read cycle_1 : Nat;
    ( /* Once Channel_1 is fulfilled it calls a sub_process Client_Event_1 to fulfill Channel_2
        [read cycle_1 == 1] -> Client_Event_1 [request, report, Channel_1, Channel_2, Channel_3]
    []
        [read cycle_1 == 0] -> Client_Event [request, report, Channel_1, Channel_2, Channel_3]
    )
process Client_Event_1 [request, report, Channel_1, Channel_2, Channel_3] : noexit :=
    Channel_2 ? read cycle_2 : Nat;
    ( /* Once Channel_2 is fulfilled it calls a sub_process Client_Event_2 to fulfill Channel_3
        [read cycle_2 == 1] -> Client_Event_2 [request, report, Channel_1, Channel_2, Channel_3]
    []
        [read cycle_2 == 0] -> Client_Event_1 [request, report, Channel_1, Channel_2, Channel_3]
    )
process Client_Event_2 [request, report, Channel_1, Channel_2, Channel_3] : noexit :=
    Channel_2 ? read cycle_2 : Nat;
    (
        [read cycle_3 == 1] -> report !1 !2 !3; /* 1.2.3 represent EPCreport respectively
    []
        [read cycle_3 == 0] -> Client_Event_2 [request, report, Channel_1, Channel_2, Channel_3]
    )
```

Specification 3 Synchronous pool mode foe fine-grained event

specification $Poll_Sync2\,[request, report, Channel_1, Channel_2, Channel_3]$: noexit
..... The main structure is the same as $Pool_Sync$.
process $ALE_Engine\,[request, report, Channel_1, Channel_2, Channel_3]$: noexit :=
$request$? $read\,cycle_1 : Nat$? $read\,cycle_2 : Nat$? $read\,cycle_3 : Nat$;
$Client_Event\,[request, report, Channel_1, Channel_2, Channel_3]$ /* Call sub_process for completing Asynchronous poll mode.
process $ALE_Client\,[request, report, Channel_1, Channel_2, Channel_3]$: noexit :=
$request$? $read\,cycle_1 : Nat$? $read\,cycle_2 : Nat$? $read\,cycle_3 : Nat$;
(
$([read\,cycle_1 == 1] \rightarrow$
$Channel_1$? g : Nat; /* get EPCs from Channel
report !g;
[]
$([read\,cycle_2 == 2] \rightarrow$
$Channel_2$? h : Nat;
report !h;
[]
$([read\,cycle_3 == 3] \rightarrow$
$Channel_3$? j : Nat;
report !j;
)

Algorithm 1: Asynchronous event cycle for poll mode

Input: The report definition and the EPCs from read cycles
Output: ECPreport
foreach *epc from* $reader_i$ **do**
  for $k = 1, \ldots, timer_i$ **do**
    Insert epc into EPCdatabase $[epc, read\,cycle_i, even\,cycle_i]$
  end
end
foreach *recorder* $[epc, read\,cycle_i, even\,cycle_i]$ *in EPCdatabase* **do**
  foreach $event\,cycle_i$ **do**
    $read\,cycle \downarrow event\,cycle_i$ = $\{read\,cycle_i$ in $read\,cycle \mid F(read\,cycle) = event_c ycle_i\}$;
    $E(report_i) \downarrow event\,cycle_i$; /* create a ALE report
    send($(event\,cycle_i, E(report_i) \downarrow event\,cycle_i)$, *ALE consumer*);
  end
end

The time efficiency of Algorithm 1 is $O(CT) + O(HC)$ where the C is number of readers, T is the duration of the timer, and H is the maximum length of EPCdatabase.

An implementation of the fine-grained event cycle for pool mode is given in two parts. The first part shown in Algorithms 2 is to send EPCs to the ALE consumer for each read cycle under event cycle, and the second part given in Algorithm 3 is to receive the EPCs and reorganize them in the EPCreport.

The time efficiency of Algorithms 2 and 3 is $O(CT) + O(CE)$ where the C is number of readers, T is the duration of the timer, and E is number of event cycles.

Algorithm 2: Forward EPCs to ALE consumer

Input: EPCs from read cycles
Output: ECPdatabase
foreach *epc from* $reader_i$ **do**
  for $k = 1, \ldots, timer_i$ **do**
    $read\ cycle_i \downarrow event\ cycle_i = \{epc \in read\ cycle_i \mid G(epc) = event\ cycle_i$;
    $E(report_i) \downarrow event\ cycle_i$; /* create a ALE report
    send($(event\ cycle_i, E(report_i) \downarrow event\ cycle_i)$, *ALE consumer*);
  end
end

Algorithm 3: Reorganize EPCreports

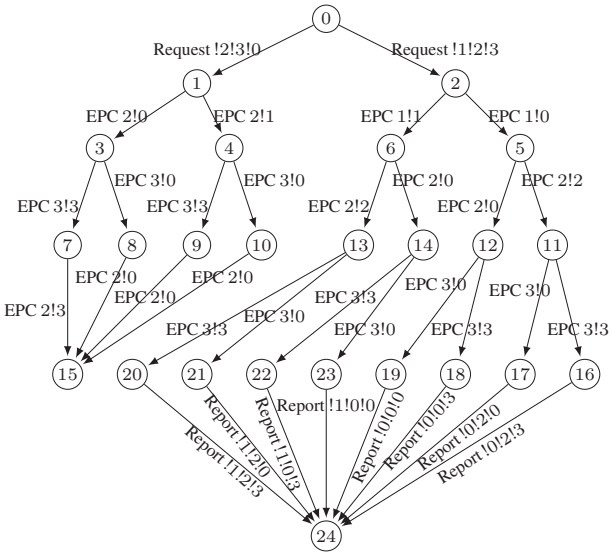
Input: EPCreport definition and all the EPCreport's
Output: ECPreport
receive($(event\ cycle_i, E(report_i) \downarrow event\ cycle_i), ALEengine$);
foreach *event* $cycle_i$ *defined by EPCreport definition* **do**
  j = get($event\ cycle_i$);
  foreach $EPCreport_i$ **do**
    $EPCreport_i \downarrow event\ cycle_j = \{EPC \in EPCreport_i \mid G(EPCreport_i) event\ cycle_i\}$;
    $E(report_j) \downarrow event\ cycle_j$; /* create ALE report for each event cycle
  end
end

IV. Experimental study

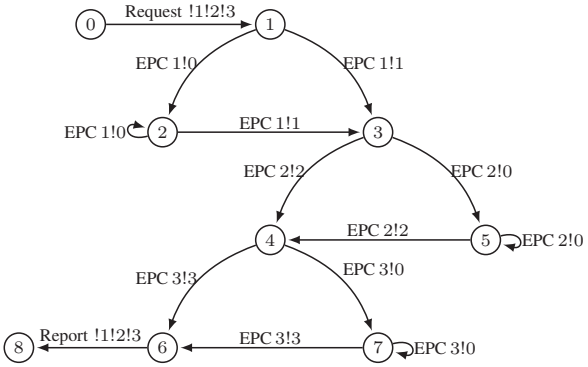
In this section we introduce two experimental simulation results, one for the synchronous mode and one for the

asynchronous mode. We used LOTOS and its tool set CADP [10] as the simulation tool. In order to evaluate our improved asynchronous poll mode we also give an example where the original synchronous poll mode generated a failed result. Our experiments were done on a Core-2 E4700 CPU with Linux OS and CADP simulation tool.

The state chart for the synchronous pool mode is given in Fig. 4a. It shows that the request !1!2!3 goes through the states $0 \rightarrow 2 \rightarrow 6 \rightarrow 13 \rightarrow 20 \rightarrow 24$ and report !1!2!3 is received at state 24. The asynchronous mode is shown in Fig. 4b.



(a) Synchronous pool mode



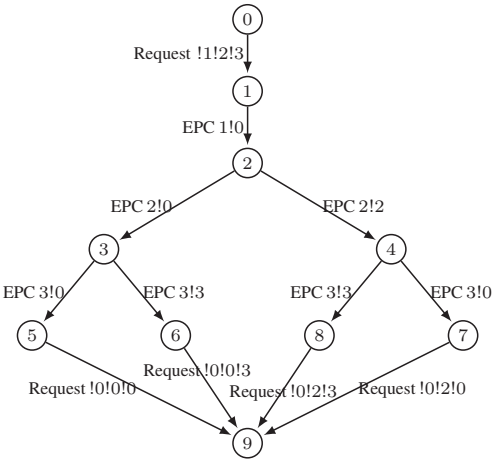
(b) Asynchronous pool mode

Fig. 4: State charts for synchronous and asynchronous poll modes

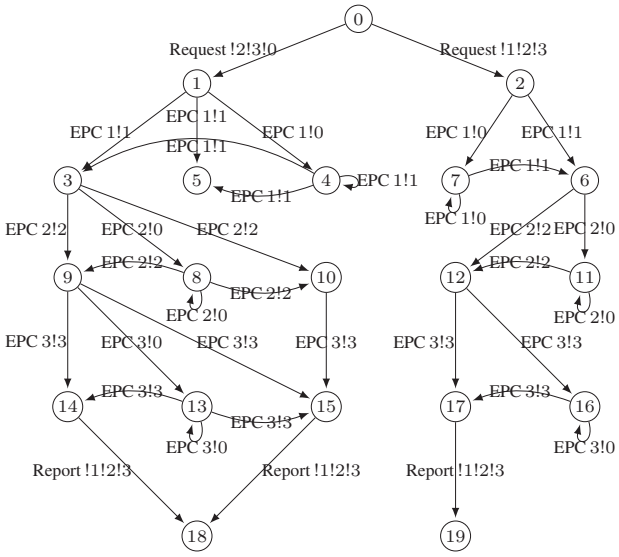
The failed example for synchronous pool mode is shown in Fig. 5a where the request does not generated a desired report. Fig. 5b shows an asynchronous pool mode experiment where the network successfully generated the requested report.

The experimental result of synchronous and asynchronous poll mode is given in Table I.

In order to check the reliability of both poll modes, we use the μ-calculus as the logic temporal language to describe the



(a) Failed synchronous case



(b) Succeeded asynchronous case

Fig. 5: State charts for failed and successful poll modes

TABLE I: Experimental result of each type of poll mode

pool mode	size of state space	number of state transitions	time (m)	Arbitrage
Synchronous	25	34	42	true
Asynchronous	9	14	48	true
Failed of synch	10	12	36	false
Success of asynch	20	37	64	true

term for successful termination. The μ-calculus express for successful term is this:

[true* . 'REQUEST !1 !2 !3' . (not 'REPORT !1 !2 !3')*]
< (not 'REPORT !1 !2 !3)* . 'REPORT !1 !2 !3' > true

That means when the ALE consumer sends Request for EPCreport then it can get replied EPCreport.

V. Related Work

One of the areas of the related work is database systems where transactions require atomicity, consistency, isolation, and durability (ACID). A centralized database management system is often used as the data repository and an arbiter that controls the execution of transactions. However, it is infeasible to impose control over scheduling of transactions at different reader over a network, and it is challenging to evaluate whether distributed transactions are conflicting.

On conflict serializability, there has been a lot of work on transaction models for mobile ad hoc networks [5], [12], [13], [14], [18] that all assumed a centralized database and an arbiter at the server, and try to address the consistency of hidden read-only transactions initiated by mobile clients [6].

Another related work are transaction management models in SOA [4]. Numerous advanced transaction models have been proposed to address the problem of providing some of the benefits of ACID transactions for long running and loosely coupled systems [7], [11]. One of such models uses a compensator to semantically undo completed operations. In this model, the system is reverted to its previous stable state when the application encounters a failure. This model has been accepted in several of the standards proposed for service oriented computing, such as BPEL4WS [1] and WSCI [2]. The drawback of this model is that the application may not want to revert to the original state in response to an exceptional event; rather it may want to handle the problem and continue making forward progress.

Recently service based transaction frameworks such as WS [3], [17], BTP [15] and WS-CAF [16] have been proposed to address the transactional problem in service oriented distributed systems. WS-Coordination defines two types of transaction protocols: WS-AtomicTransaction and WS-BusinessActivity. A detailed descriptions of these protocols can be found in [11]. BTP and WS-CAF also provide a set of patterns and protocols, but do not deal directly with the problem of consistency and isolation.

VI. Conclusion

With the rapidly development of EPC Network the user have turned attention to the reliability and the performance of application system. As a key component of EPC framework, ALE plays an important role in reliability and performance in EPC applied systems. In this paper we proposed two methods for improving ALE's reliability and performance. By adapting original poll synchronous mode to the new asynchronous mode, the reliability of ALE is enhanced; and by separating fine-grained cycle from the coarse event cycle, EPC network channel can be released to prevent traffic block. The simulation results has shown the effectiveness of the proposed methods.

We are now developing a real EPC network applied system for a retail enterprise. In fact, part of the experimental study was from this real EPC system. We will continue to complete the ALE implementation with the idea presented in this paper.

Acknowledgment

This work was partially supported by National Natural Science Foundation of China (Grant No. 61070030), Innovative Scientific Research Team of Beijing Education Committee (Grand No. 4062012), and Beijing Government and Education Committee (Grant No. PHR201107107).

References

[1] Tony Andrews and et. al. Business process execution language for Web Services, version 1.1. http://public.dhe.ibm.com/software/dw/specs/ws-bpel/ws-bpel.pdf, 2003.

[2] Assaf Arkin and et. al. Web service choreography interface (WSCI) 1.0. http://www.w3.org/TR/wsci, 2002.

[3] Luis F. Cabrera, George Copeland, Jim Johnson, and David Langworthy. Coordinating Web services activities with WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity. 2004.

[4] Patrick F. Carey and Bernard W. Gleason. Solving the integration issue – service-oriented architecture (SOA). *Executive White Paper Series*, 2005.

[5] Il Young Chung, Bharat Bhargava, Malika Mahoui, and Leszek Lilien. Autonomous transaction processing using data dependency in mobile environments. In *Proceedings of the 9th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 138–144, 2003.

[6] Murat Demirbas, Onur Soysal, and Muzammil Hussain. TRANSACT: A transactional framework for programming wireless sensor/actor networks. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks*, pages 295–306, 2008.

[7] Ahmed K. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[8] EPCglobal. The application level events (ALE) specification, version 1.0. http://www.epcglobalinc.org/standards, 2005.

[9] EPCglobal. Architecture framework final version. http://www.epcglobalinc.org/standards/architecture/architecture_1_0-framework-20050701.pdf, 2005.

[10] Hubert Garavel, Radu Mateescu, Frédéric Lang, , and Wendelin Serwe. CADP 2006: A toolbox for the construction and analysis of distributed processes. In *Proceedings of the 19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163. Springer, 2007.

[11] Paul Greenfield, Dean Kuo, Surya Nepal, and Alan Fekete. Consistency for web services applications. In *Proceedings of the 31st international conference on Very Large Data Bases*, pages 1199–1203. ACM, 2005.

[12] Kam-Yiu Lam, Mei-Wai Au, and Edward Chan. Broadcast of consistent data to read-only transactions from mobile clients. In *Proceedings of the 2nd IEEE Workshop on Mobile Computer Systems and Applications*, pages 80–88, 1999.

[13] Victor C. S. Lee and Kwok-Wa Lam. Optimistic concurrency control in broadcast environments: Looking forward at the server and backward at the clients. In *Proceedings of the First International Conference on Mobile Data Access*, MDA '99, pages 97–106. Springer-Verlag, 1999.

[14] Victor C. S. Lee, Kwok-Wa Lam, Sang H. Son, and Eddie Y. M. Chan. On transaction processing with partial validation and timestamp ordering in mobile broadcast environments. *IEEE Transactions on Computers*, 51(10):1196–1211, 2002.

[15] OASIS Business Transactions Technical Committee. Business transaction protocol, version 1.0. http://www.oasis-open.org/committees/download.php/1184/2002-06-03.BTP_cttee_spec_1.0.pdf, 2002.

[16] OASIS Web Services Composite Application Framework (WS-CAF) Technical Committee. Web services context specification (WS-context), version 1.0. http://docs.oasis-open.org/ws-caf/ws-context/v1.0/OS/wsctx.pdf, 2007.

[17] OASIS Web Services Transaction (WS-TX) Tech Committee. Web services coordination (WS-coordination), version 1.2. http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os.pdf, 2009.

[18] Jayavel Shanmugasundaram, Arvind Nithrakashyap, and Rajendran Sivasankaran. Efficient concurrency control for broadcast environments. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 85–96. ACM, 1999.

[19] Peter Van Eijk and Michel Diaz. *Formal Description Technique Lotos: Results of the Esprit Sedos Project*. Elsevier Science Inc., 1989.