Scheduling of Concurrent Transactions in Broadcasting Environment

Ahmad Al-Qerem[1], Ala Hamarsheh[2], Yaser A. Al-Lahham[3] and Mujahed Eleyat[4]
[1]Department of CIS, Zarqa University, Zarqa, Jordan
[e-mail: ahmad_qerm@zu.edu.jo]
[2]Faculty of Engineering and Information Technology, Arab American University, Jenin, Palestine
[e-mail: ala.hamarsheh@aauj.edu, alaauj@gmail.com]
[3]Department of Computer Science, Zarqa University, Zarqa, Jordan
[e-mail: yasirlhm@zu.edu.jo]
[4]Faculty of Engineering and Information Technology, Arab American University, Jenin, Palestine
[e-mail: mujahed.eleyat@aauj.edu]
*Corresponding author: Ala Hamarsheh

Received June 3, 2017; revised October 2, 2017; accepted December 4, 2017; published April 30, 2018

Abstract

Mobile computing environment is subject to the constraints of bounded network bandwidth, frequently encountered disconnections, insufficient battery power, and system asymmetry. To meet these constraints and to gain high scalability, data broadcasting has been proposed on data transmission techniques. However, updates made to the database in any broadcast cycle are deferred to the next cycle in order to appear to mobile clients with lower data currency. The main goal of this paper is to enhance the transaction performance processing and database currency. The main approach involves decomposing the main broadcast cycle into a number of sub-cycles, where data items are broadcasted as they were originally sequenced in the main cycle while appearing in the most current versions. A concurrency control method AOCCRBSC is proposed to cope well with the cycle decomposition. The proposed method exploits predeclaration and adapts the AOCCRB method by customizing prefetching, back-off, and partial backward and forward validation techniques. As a result, more than one of the conflicting transactions is allowed to commit at the server in the same broadcast cycle which empowers the processing of both update and read-only transactions and improves data currency.

Keywords: mobile computing, data broadcasting, concurrency control

1. Introduction

Mobile computing has become remarkably used by a wide range of users; as a consequence of the vast improvement in computer hardware and wireless network technologies. This environment encourages many providers to offer services to an unbounded number of users who are equipped with portable computing devices running on battery power regardless of time and distance limitations.

Existing mobile technologies suffer from bounded network bandwidth, frequently encountered disconnections, insufficient battery power, and system asymmetry constraints. To handle such limitations, wireless data broadcasting has been applied on data transmission techniques by many research efforts [1][2][3][4][5][6].

Generally, in mobile client-server architecture, servers respond to users' requests while requests are being transmitted, mobile clients consume more of their power than that when they receive the response in addition to the consumption of the scarce uplink bandwidth. Since the number of requests from any given client is not bounded as of the case of the number of the requesting clients, the server may be heavily loaded resulting in an extremely increased response time. However, wireless data broadcasting models such as "Broadcast Disks Model", which was proposed by Acharya [1], which can overcome these problems. In this model, the server constantly broadcasts the entire database via one or more wireless communication channels. Data items of interest to any client are retrieved by that client when they come up on the channel. So, clients have to wait for the required items until they are in the channel. This system identifies the channel as a shared data repository (disk); where data items are accessed sequentially. As broadcasting any item can satisfy all the outstanding requests for that item simultaneously, whatever the number of the outstanding requests is, the access time of mobile clients is not affected. Wireless data broadcasting is impressively scalable; that's why it is widely used to develop several mobile application systems, like: auctions, electronic bidding, stock trading, weather information and traffic information broadcasts [7].

Read-only transactions constitute the majority in these applications while update transactions are infrequent. For example, stock trading involves a small group of stock purchasing or bidding, which represents update transactions in the application. Besides, relatively many more brokers, who just monitor stock prices, issue read-only transactions. In a read-only condition, there is no concerns about the consistency among data items, while in the presence of update transactions, consistency is most likely to be violated [6][8][9][10].

Reasonably, concurrency control schemes are required for mobile transactions to preserve data currency and consistency. Nevertheless, a direct application of traditional concurrency control schemes to mobile transaction processing is not considered an option because they do not suit the limitations of this environment [8]. In conventional methods, the communications between mobile clients and the server comprise exchanging a large number of messages, which in return consume much battery power of mobile clients and the limited uplink bandwidth [11], [12][13].

Moreover, the unbounded number of transactions coming from an also unbounded number of clients can easily overload or interrupt servers. Consequently, traditional concurrency control schemes, based on locking and time stamping, are not suitable for mobile transaction processing [7].

This paper proposes a new data, which addresses a technique that enhances the concurrent usage of a wireless database transmission. The new technique decomposes the cycle into subsycles allowing more than one conflicting transaction to be committed in a single global

cycle, and that is because an aborted transaction of one subcyle is shifted to resume execution at the next subcycle. The rest of the paper is organized as follows.

2. Related Work

Most of research efforts in wireless broadcasting circumstances concurrency control revolved around mobile transactions with uniform data access patterns [15], [16], [6], [8], [17], [18], [9], [10]. For example, Lee [8] proposed a method, which is called "Forward and Backward Optimistic Concurrency Control" denoted by "FBOCC", based on the optimistic concurrency control method.

Forward validation is assigned for update transactions, while partial backward validation is for read-only transactions in FBOCC [8]. Nevertheless, just like all other concurrency control methods for wireless broadcast environments; FBOCC concentrates on mobile transactions with uniform data access patterns. When mobile clients run update transactions with non-uniform data access patterns, all those methods show a poor performance because of the frequent aborts and restarts in the final validation phase, which is a result of the conflict caused by updating the same data items. Accordingly, this problem wastes both the uplink and the downlink bandwidth as well, which in turn wastefully consumes more of the battery power of mobile clients. Later mobile concurrency control methods did not exploit caches of mobile clients. Caches at mobile clients decrease their response time, because storing data items in the mobile cache enables them to be accessed locally when the relevant transactions are resumed.

For the sake of making the system more responsive, many proposals were deployed. Examples of these proposals are investigated, Lee [17][18] proposed a predeclaration technique, such that the response time of restarting mobile transactions is decreased by prefetching the data items before that transaction starts. Prefetching can be performed either through declaring the read data set of a transaction when it starts, or searching the transaction for all the potentially required data items before it starts. Nevertheless, the first method is impractical, while the second used more resources are consumed since more items are read more than what is really needed [7].

Multiversion data broadcast technique [19] maintains and broadcasts multiple versions for each item instead of broadcasting the last committed version only. The main purpose is to enhance the commitment probability. A new version (holding the identifier of the desired cycle) is assigned to each data item at the beginning of each cycle. The clients are supposed to access the different data items of the same version. The multiversion broadcast method neither supports update transactions nor real time conditions. Likewise, the size of the broadcast cycle is the added item versions, which, in turn, increase the response time for mobile transactions.

Adaptive optimistic concurrency control with random back-off (AOCCRB) method was proposed on the basis of the optimistic concurrency control scheme. Optimistic concurrency control methods allow transactions to run their operations and defer the validation phase assuming that conflicts hopefully will never occur [7]. The validation is carried out in two modes, forward validation and backward validation [12]. In backward validation, the transaction checks the data items stored in its local buffer for reading purposes, against the control information (CI) attached at the beginning of each broadcast cycle. If a conflict is found, the validating transaction aborts locally. Hence, the control information of a cycle holds all the data items that were updated at the server by committed transactions in the previous cycle [7]. Forward validation involves only update transactions, when the validating transaction submits the set of data items two steps are incorporated. In the first step, the server

carries out a backward validation against the transactions that were committed since the start of the current cycle and before the data set is submitted. If the validating transaction does not pass the first step, it aborts. In the second step, the validation is carried out against the currently running transactions. If conflictions are detected, they will be resolved by aborting the currently conflict running transactions. Reasonably, aborting running transactions is preferable because of the relatively high loss of sacrificing the validating transaction [7].

To improve the response time of mobile transactions, AOCCRB applies data prefetching technique based on the access invariance [20]. Access invariance denotes that a restarting transaction heads for requiring the same set of data items used in the previous failed execution. A mobile client maintains a list of the data items accessed when it is running a transaction, so when the transaction is aborted and restarted, the data list is prefetched into the clients' cache, enhancing the response time of mobile transactions by reducing the access time for the needed data items. As mobile update transactions have non-uniform data access patterns, they are potentially repeatedly aborted and restart because of the update conflicts on hot data items, wastefully consuming resources of mobile clients [7]. AOCCRB supports mobile and server update transactions, real-time environment, and implements serializability as a correctness criterion. However, only one transaction, among all the conflicting transactions, is allowed to commit in any cycle. Consequently, the waiting time for restarting update transactions could be significantly long due to the size of the cycle. Furthermore, the availability of the updated items in any cycle is delayed to the next cycle.

STUBcast proposed by Huang in [21], introduced two novel correctness criteria, "single serializability" and "local serializability" denoted by "SS"and "LS", respectively. Single serializability asserts that "all update transactions and any single read-only transaction, are serializable". Whilst, local serializability requires "all the update transactions in the system and all read-only transactions at one client side to be serializable". SS and LS are weaker but easier to achieve than global serializability, and they do guarantee the consistency and correctness at the server database. In STUBcast broadcast operations are divided into "primary broadcast" and "update broadcast" denoted by "pcast" and "ucast", respectively. The pcast broadcasts all the data items with their versions prior to the beginning of dissemination, while ucast broadcasts the new versions of the updated data items by inserting them into the ongoing pcast whenever a transaction is newly committed. The protocol consists of three components, "Client Side Read only Serialization Protocol" denoted by "RSP", "Client Side Update Tracking and Verification Protocol" denoted by "UTVP", and "Server Side Verification Protocol" dented by "SVP".

STUBcast supports only mobile client update transactions, and does not support real-time environment, it uses single and local serializability (which is weaker than serializability) as a correctness criterion. Furthermore, the access efficiency is affected by the extended length of the cycle, besides the high overhead from using timestamps and the complexity of computations.

3. System Model

This section presents the model of the proposed system, which includes the following items: system architecture, structure and formation of the broadcast, data scheduling, control information creation and placement, and data binding (i.e. when data items are extracted from the database). These are all customized to achieve the desired targets. Moreover, some new assumptions and modified constraints will be presented.

3.1 Subcycle Formation Procedure

The ordinary broadcast cycle approach broadcasts a subset of the database (or the whole database) as a transmission cycle. The broadcast subset consists of a subset of a database that includes all the data items aimed for dissemination. Broadcast cycle items are disseminated through a broadcasting channel in the form of a sequential data stream. The proposed model creates the conventional broadcast cycle, in the form of a set of data items only, then it splits it into smaller intervals called "sub-cycles". Creating *AOCCRBSC* adaptive cycle is carried out as follows:

1. All cycles are of the same fixed size (i.e. each cycle contains the same number of items), as in **Fig. 1**(a).

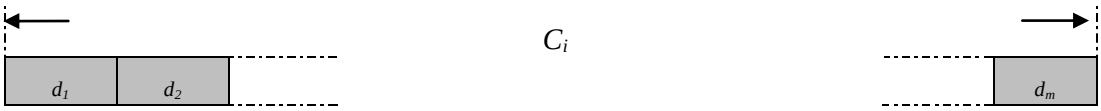

Fig. 1(a). Original Broadcast Cycle

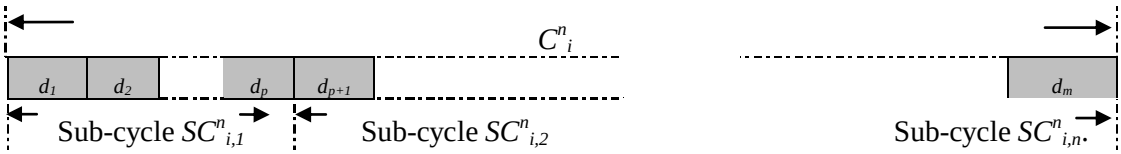

Fig. 1(b). Decomposed broadcast cycle

2. Sub-cycles are generated by splitting the cycle into none overlapping equally sized partitions that contain all the data items included in the original cycle, as shown in **Fig. 1(b)**. Throughout this model the notation: C^n_i refers to the ith cycle that consists of 'n' sub-cycles. Let D_m= {d_1, d_2, …, d_m}, be the set of data items included in C^n_i. As sub-cycles are equally sized and none overlapping, 'n' is a divisor of 'm' and $SC^n_{i,j}$ is the jth sub-cycle in the ith cycle. So given that $SC^n_{i,j}$ ={d_p}, where: $d_p \in D$, implies that ((j-1)*(m/n)+1) < p <= (j*m/n) where 'p' is the sequence of the data item 'd' in the D_m (the same as in the original cycle before decomposition), and $j \in \{1, 2, \ldots, n\}$. The pseudocode for cycle decomposition is shown in **Fig. 2**.
3. All cycles consist of the same number of sub-cycles.
4. The data items are extracted from the database dynamically. In other words, when the server starts for the first time, the first generated broadcast cycle consists only the identifiers of the data items selected for dissemination, the cycle is partitioned into subcycles. When it is time to start extracting data items from the database, only the items of the first sub-cycle are extracted, other items are postponed until the time of their dedicated subcycle. The server repeats these steps for all of the subsequent cycles.

CycleDecompse(D, n) { for j = 1 to n for p = ((j - 1) * (m / n) + 1) to ((j * m / n)) $DD_j = DD_j + d_p$; endfor endfor }	\\$D$: the set of data items to be used to generate the adaptive cycle \\$m$: the number of items included in D \\$d_p$: the data item with sequence number 'p' in D \\$n$: the number of sub-cycles desired \\$DD_j$: the jth division of D

Fig. 2. Cycle Decomposition

5. Indexing of the data items in a cycle is distributed over its sub-cycles so that each data item in the decomposed cycle will be in the same order it appeared in the original cycle. The index is attached at the beginning of each sub-cycle as shown in **Fig. 3**. The index of each sub-cycle contains: the id of the data item, paired with the time when it will appear, the id of all sub-cycles it will appear in, paired with the subcycle start. And finally, the id of the next cycle paired with its start time. Time is measured relative to the starting time of the sub-cycle where the index is attached, for example; suppose that the pairs: (C_{i+1},100), ($SC^n_{i,j+1}$,21) and (d_k,15) be a subset of information in the index of the current sub-cycle "$SC^n_{i,j}$". Each pair is interpreted as follows: the first, indicates that the i^{th} cycle C_i will start after 100 time units from the time that $SC^n_{i,j}$ has started. The second, tells that the sub-cycle number (j+1) in the ith cycle (hence, the current cycle) will appear after 21 time units also from the time that $SC^n_{i,j}$ has started. And the last, shows that the k^{th} data item d_k will appear after 15 time units also from the time that $SC^n_{i,j}$ has started.

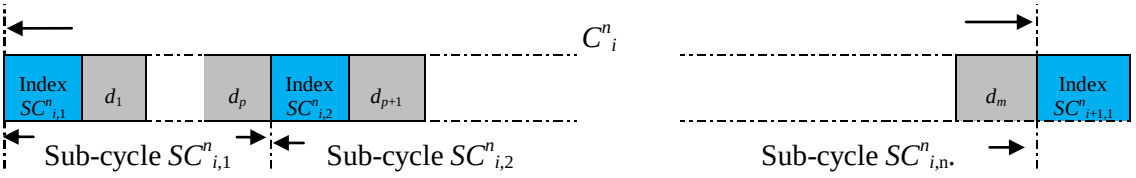

Fig. 3. Decomposed broadcast cycle C^n_i with index

6. In the ordinary cycle, control data is cached at the beginning of each new cycle. Control information of a cycle includes all the data items that were updated at the server in the previous cycle. Control information is used to enable the running mobile transactions to detect the read/write conflicts with the committed transactions recorded on the server in the previous cycle, so they abort early saving valuable resources. Control information can be used to detect these conflicts coming from the previous sub-cycle, such that control information at the beginning of each sub-cycle includes all the data items that were updated at the server in the last sub-cycle, as shown in **Fig. 4**. Consequently, running mobile transactions have to suspend execution at the beginning of every new sub-cycle to catch the control information, for validation.

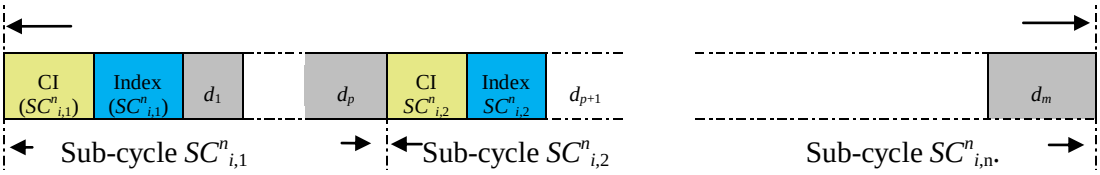

Fig. 4. Decomposed broadcast cycle C^n_i with index and Control Information (CI)

As shown in **Fig. 4**, transactions do not have to wait until the length of the reminder of cycle to catch CI and execute the validation, instead it waits for the rest of current subcycle, giving a chance to more transactions to be committed, and conflicts may be detected earlier. Number of sub-cycles generated has a big impact on the performance, so it should be chosen carefully. Higher number of sub-cycles generated, the faster CIs appear. Consequently, running transactions must tune-in more often in a shorter time, since the tuning time may dramatically increase. For example, if more subcycles are used, the time spent in doze mode before tuning-in should be considered when determining the number of sub-cycles in order to not missing any control information (CI) by transactions.

3.2 Adaptive Optimistic Concurrency Control at Sub-Cycle Level (AOCCRBSC)

Broadcast cycle is a subset of a consistent state of the database, so only transactions that gain all its data items, and finish execution within the same cycle are allowed to commit, while the rest, are aborted and restarted later at the beginning of the next cycle. Consequently, transactions are forced to abort until insuring that data items collected in different broadcast cycles are members of a consistent state of the database.

Backward validation technique solves the problem of uncertainty and discovers inconsistency by allowing running transactions to span multiple cycles detecting the inconsistency of the data items they read so far, by invalidating data items in its read set at the beginning of every broadcast cycle against all the data items updated by the transactions that were committed during the last cycle.

Unfortunately, once a conflict is detected, not only the transaction has to restart, but also has to read the entire data items it already read, including none conflicting items. This situation needs more resources since the transaction is forced to read data items from air instead of reading it from its own cache. To solve this problem, a restarting transaction should re-read only the inconsistent data items from the broadcast cycle, and caching the consistent items. This is what prefetching stands for.

Combining prefetching and backward validation will save mobile client's resources, since the number of successfully committed transactions is increased, and the number of data items read from the air is decreased.

The server has to validate update transactions for commit, or to do what is called "Forward Validation". Forward validation completes the work done by the backward validation at the mobile client. In forward validation, the intending to validate transaction is checked out against all active transactions during the current sub-cycle. However, aborting the validating transaction is more expensive than aborting the currently running transactions, the resolution of any potential conflicts is carried out by aborting the currently conflict running transactions.

Basically, random back-off decreases the number of restarting transactions at the beginning of a new cycle, by trying to decrease the number of aborts. Assume that there are k transactions conflicting because of data item '*x*'. When any of them runs the forward validation, the rest *k*-1 transactions are aborted and restarted at the next cycle. Thus, the

opportunity for any of the transactions to commit is (1/*k*). This case is repeated for at least *k*-1 cycles, because new competitors may appear in next cycles. The basic idea of the random back-off is that the server keeps track of the number of transactions that aborted because of their intentions to update a data item, which is called the contention degree of the data item '*x*'. Including this parameter in the control data, and transmit it to the mobile client enables it to postpone aborted transactions of low committing possibility (i.e. with the high contention degree) to the start of a later cycle.

Consequently, combining all of these techniques (prefetching, backward validation, forward validation, and random back-off) will be more significant. However, when broadcast cycles are too long, some aborted transactions will suffer from long delay. If transactions have time-outs, as it is so often the case, some transaction will not survive. Dead transactions have to be re-submitted for execution, which makes the mobile clients to hold for longer time. Therefore, degrading the data currency because the server will spend more than one cycle to validate data items to consistently appear in the next cycle.

Allowing a subset of data items to be updated at the server during the current cycle, data items will always appear in their most updated versions (i.e. increasing data currency), allowing more than one transaction to update the same data item during one cycle (i.e. increasing transaction concurrency), at the same time reducing resource consumption by decreasing number of data items read from air by restarted transactions, while applying the undisputed correctness criterion (i.e. the serializability), so that the consistency is still preserved at once. This can be assumed to be a sensational achievement, which is the contribution of this work.

3.2.1 Partial Backward Validation at Sub-Cycle Level at Mobile Client

This section presents an explanation of how transactions work at mobile client side in the proposed method; mobile transactions are processed throughout the following steps:

1. The local cache of a mobile transaction is utilized so that any read operation checks for the existence of the desired data item before trying to fetch it from air. In addition, all write operations initially take place in the local cache until after the transactions finish execution, that they are submitted to the server for validation. Furthermore, a restarting transaction predeclares the data items to be read from air again, such that data items are ordered according to its sequence in the channel instead of the time they were required in the execution.
2. Whenever a new sub-cycle begins, all transactions suspend execution and perform a partial backward validation using the broadcasted control information. The transactions which succeed the validation resume the execution, other failure transactions are aborted. The time, when an aborted mobile transaction restarts, depends on the type of the transaction (i.e. read-only transactions immediately restarted, while update transactions wait for the back-off time before restarting.
3. When a read-only transaction reaches its end of transaction point (EOT) before a new broadcast sub-cycle, it locally commits; otherwise, if it is an update transaction, it goes through the forward validation procedure at the server.

Step1 extends the behavior of the existing optimistic concurrency control method by enhancing the utilization of the local cache of restarting transactions. In addition to prefetching, which already exists, a restarting transaction exploits cache for predeclaration by sorting the unprefetched data items by their index in order to catch each of them from its earliest occurrence.

In step 2, instead of waiting the next cycle, running mobile transactions (both read-only and update transactions) suspend execution to catch the CI, which contains only the items that were updated during the previous sub-cycle, to run a partial backward validation.

In step 3, as read-only transaction locally commit, the updated data items will be available in the channel immediately at the first sub-cycle they are scheduled to before the beginning of the whole cycle.

The Algorithm for Partial Backward Validation at Sub-Cycle Level

Fig. 5 represents partial backward validation at sub-cycle level. When the j^{th} sub-cycle of the i^{th} cycle, which consists of n sub-cycles, denoted by "$SC^n_{i,j}$" starts, the mobile transaction 'T_v' suspends running to go through the partial backward validation. The validating transaction detects the channel to catch the control information broadcasted at the beginning of the sub-cycle. Afterwards, it compares the set of data items in the captured control information against the cached items at T_v for reading purposes creating its "ReadSet(T_v)".

```
//Tv: mobile transaction.
//CI(SCi,j): the set of data items that was updated during the previous broadcast //sub-cycle 'SCx,y'.
//ReadSet(Tv): the read data set of the transaction 'Tv'
//PrefetchingSet(Tv): the set of data items to be prefetched by the transaction 'Tv', //i.e., to be read from the
local cache.
//UnPrefetchSet(Tv): the set of data items to be unprefetched by Tv, i.e., to be read //from the air again.
PartialBackwardValidationSC(Tv, SCni,j)
{
if(ReadSet(Tv) ≠ Ø)
  forall 'x' ϵ CI(SCni,j)
    forall 'y' ϵ ReadSet(Tv)
      if (x.id == y.id) then
        UnPrefetchSet(Tv) = UnPrefetchSet(Tv) +y;
      endif
    endfor
  endfor
endif
if(j>1) then
  C = SCni,j-1;
else
  C = SCni-1,n;
endif
if(UnPrefetchSet(Tv) ≠ Ø) then
  PrefetchingSet(Tv) = PrefetchingSet(Tv) U (ReadSet(Tv) -
  UnPrefetchSet(Tv));
  if (Tv is update transaction) then
    BackoffAndRestartSC(Tv, C);
  else
    restart(Tv);
  endif
else
  continue;
  when Tv finishes, call LastPartialBackwardValidationSC(Tv, SCni,j);
endif
}
```

Fig. 5. Partial Backward Validation at Sub-Cycle level

Unless there are no common items resulting from this comparison, T_v aborts locally due to confliction with a committed transaction in a previous sub-cycle, otherwise it resumes running.

Before the transaction 'T_v' is aborted, it extracts the non-conflicting data items from the set of data items included in ReadSet(T_v), adding them to the prefetching set denoted by "PrefetchingSet(T_v)". T_v will reread the prefetched data items, when it restarts, locally from its cache. Furthermore, conflicting data items are also extracted, and included in another set denoted by "UnPrefetchSet(T_v)". T_v will reread the unprefetched data items from air again, sorting them according to the arrival time (index), and storing them in its local cache as soon as they show up in the air.

If 'T_v' is an update transaction, the abortion will trigger algorithm2 which is used to determine the back-off time that T_v should spend waiting before restarting. Otherwise, if 'T_v' is a read-only transaction, it is just restarted normally.

```
//Tv: mobile transaction.
//CI(SCi,j): the set of data items that was updated during the previous broadcast sub-cycle 'SCx,y'.
//ReadSet(Tv): the read data set of the transaction'Tv'
//WriteSet(Tv): the current write data set of the transaction 'Tv'
//PrefetchingSet(Tv): the set of data items to be prefetched by the transaction 'Tv', //i.e., to be read from
the local cache.
//UnPrefetchSet(Tv): the set of data items to be unprefetched by Tv, i.e., to be read // again from air.
//ContentionDegree[x]: the contention degree of the data item 'x'
BackoffAndRestartSC(Tv, SC^n_i,j){
abort(Tv);
MaxContentionDegree = max(ContentionDegree[x]) where x ϵ ReadSet(Tv);
if(MaxContentionDegree>1) then
MaxContentionDegree = MaxContentionDegree – 2;
endif
Pick a random number between 0 and (MaxContentionDegree) as BackoffTime;
k = 1;
while(k <= BackoffTime)
  wait in doze mode until the next sub-cycle starts;
  forall 'x' ϵ CI(SC^n_i,j) do
    forall 'y' ϵ PrefetchingSet(Tv) do
      if(x.id == y.id) then
        UnPrefetchSet(Tv) = UnPrefetchSet(Tv) + y;
      endif
    endfor
  endfor
  PrefetchingSet(Tv) = PrefetchingSet(Tv) - UnPrefetchSet(Tv);
  k = k + 1;
end while
sort(UnPrefetchSet(Tv)); // according to the index of the items contained
restart(Tv);
}
```

Fig. 6. Back-off and Restart at Sub-Cycle level

Back-Off and Restart at Sub-Cycle Level

Fig. 6 shows the pseudo code for back-off and restart at sub-cycle level, where ($SC^n_{p,q}$) is the sub-cycle leading the current sub-cycle ($SC^n_{i,j}$), the transaction 'T_v' is first aborted. The

mobile client finds the data item having maximum contention value M in the ReadSet(T_v), which is broadcasted at the beginning of the sub-cycle. After that, the random number, denoted by "BackoffTime", is generated depending on the value of M. Finally, T_v waits in doze mode for BackoffTime cycles before restarting.

Mobile clients, including those in back-off phase, get the control information at the beginning of each sub-cycle. If it finds out that one of the locally cached data items is updated in the previous sub-cycle, the items are excluded from PrefetchingSet(T_v) and included in UnPrefetchSet(T_v). If the transaction 'T_v' successfully goes through the partial backward validation, it resumes running.

Last Partial Backward Validation at Sub-Cycle Level

Fig. 7 shows the pseudo code of the last partial backward validation at sub-cycle level. When a transaction 'T_v' finishes execution before the beginning of a new sub-cycle, it is locally committed if it is a read only transaction, otherwise it waits for server validation. The client stores the current sub-cycle number, denoted by "$SC^n_{i,j}$", in its cache, and submits it to the server after the transaction T_v finishes execution, where is the server use it for the final validation.

```
//Tv: mobile transaction
//CI(SCi,j): the set of data items that was updated during the previous broadcast sub-cycle 'SCx,y'.
//ReadSet(Tv): the current read data set of the transaction 'Tv'
//WriteSet(Tv): the current write data set of the transaction 'Tv'
//PrefetchingSet(Tv): the set of data items to be prefetched by the transaction 'Tv'
LastPartialBackwardValidationSC(Tv, SC^n_i,j)
{
PrefetchingSet(Tv) = PrefetchingSet(Tv) U ReadSet(Tv);
if(Tv is read-only transaction) then
  commit(Tv);
  exit();
endif
if(j>1) then
  C = SC^n_i,j-1;
else
  C = SC^n_i-1,n;
endif
submit WriteSet(Tv) and C to server;
if(Tv is validated) then
  commit(Tv);
  exit();
else
  BackoffAndRestartSC(Tv, C);
endif
}
```

Fig. 7. Last Partial Backward Validation at Sub-Cycle level

3.2.2 Forward Validation at Sub-Cycle Level at the Server

The server does forward validation by comparing the write data set of the validating transaction against the read data sets of all currently running transactions. Transactions running on the server having common items between their read data sets and the write data

set of the validating transaction, making it to abort. This is the way in which data conflicts are resolved.

Passing forward validation by a mobile transaction is constrained to passing a preceding last backward validation. This is reasonably required because, during the current sub-cycle, specifically after the transaction has locally passed the partial backward validation and before the transaction went through final validation, some transactions may be committed at the server. Accordingly, the mobile client records the corresponding sub-cycle number during which the partial backward validation is passed, and it submits it and the desired write data set to the server for forward validation.

```
//Tv: the validating mobile transaction
//Ta: a set of the currently running server transactions
//SCx,y: the broadcast sub-cycle number when 'Tv' passed the last partial backward //validation
//Tc : a set of transactions which committed at the server during the broadcast //sub-cycle SCx,y
//ReadSet(T): read data set of the transaction 'T'
//WriteSet(T): write data set of the transaction 'T'
//CI(SCx,y): the set of the data items that was updated during the previous //broadcast sub-cycle 'SCx,y'
//ContentionDegree[x]: the contention degree of the data item 'x'
ForwardValidationSC(Tv, SC^n_i,j){
if(Tv is a mobile transaction) then
  if(LastBackwardValidationSC(Tv) is false) then
    abort(Tv);
    forall 'x' ϵ WriteSet(Tv) do
      ContetionDegree[x] = ContetionDegree[x] + 1;
    endfor
  endif
endif
forall T ϵ Ta do
  if(Readset(T) ∩ WriteSet(Tv) ≠ Ø) then
    abort(T);
  endif
endfor
if(j<n) then
  CI(SC^n_i,j+1) = CI(SC^n_i,j) U WriteSet(Tv);
else
  CI(SC^n_i+1,1) = CI(SC^n_i,j) U WriteSet(Tv);
endfor
commit(Tv);}
LastBackwardValidationSC(Tv)
{
forall T ϵ Tc do
  if(WriteSet(T) ∩ ReadSet(Tv) ≠ Ø) then
    return false;
  endif
endfor
}
```

Fig. 8. Forward Validation at Sub-Cycle level

The Algorithm for Forward Validation at Sub-Cycle Level

Fig. 8 shows the pseudo code for forward validation at sub-cycle level. Whenever a transaction is committed at the server, the database is updated using the write data set of that

transaction. The write data sets of all committed transactions in a sub-cycle are maintained by the server to generate the control information that will be broadcasted at the beginning of the next broadcast sub-cycle. Control information is used by mobile clients to run the partial backward validation for their mobile transactions. If the validating transaction 'T_v' fails the at forward validation, the server exploits the write data set of 'T_v' to calculate the contention degree of each of these items. Finally, successfully committed update transaction is adding to the control information of the next sub-cycle, as a result that transaction commits before it terminates.

Adjusting Contention Degree at Sub-Cycle Level

Algorithm 6 shows the pseudo code for adjusting contention degree at sub-cycle level. At the beginning of each sub-cycle, the contention degrees of all data items are readjusted by using "Algorithm5". If there were no transactions competing to write on the item '*x*', then the contention degree of '*x*' is initialized as (0). If there were '*k*' update transactions competing to write on '*x*', the aborted transactions will pick a random number 'w' between (0) and (*k*-2) and have to wait until 'w' sub-cycles are elapsed before they are restarted. Due to this method, the '*k*' update transactions competing for the data item '*x*' are expected to be uniformly distributed across the next '*k*' broadcast cycles. Thus, the contention degree of the data item '*x*' in this case is readjusted to (1).

```
//ContentionDegree[x]: the contention degree of the data item 'x'
AdjustContentionDegreeSC()
{
forall 'x' in DB do
  if(ContentionDegree[x] > 1) then  ContentionDegree[x] = 1;
  else   ContentionDegree[x] = 0;
  endif
endfor
}
```

Fig. 9. Adjusting Contention Degree at Sub-Cycle level

4. Performance Analysis

In this section, the performance of the proposed method AOCCRBSC is analyzed using the experimental results from the simulator, which is implemented regarding the adaptive model against AOCCRB [7]. Testing was carried out through out a repetitive comparison between the set of results coming from the simulator and a set of manually produced results applying the same conditions (i.e. parameter settings). In the next sections (5.1, 5.2), the simulator is described, the simulation results are analyzed and discussed, respectively.

4.1 Simulator Specifications

The simulation program is built to evaluate the performance of the proposed AOCCRBSC (Adaptive Optimistic Concurrency Control using Random Back-off at Sub-Cycle level) scheme against AOCCRB [7]. This simulator is an object-oriented program which is implemented using Java programming language running on a 1.00 GHz AMD E1-2100 APU processor PC equipped with 4 GB of RAM and using Windows 7 Ultimate SB1 platform.

Mobile transactions in the simulator are allowed to keep running until they eventually commit, regardless of any time limitations. In other words, the simulated environment does

not support real-time conditions; so that a mobile transaction does not have to meet any additional deadline constraints to commit. The parameter settings of the simulator are shown in **Table 1** down below.

Table 1. Simulation Parameters

Component	Parameter	Range
Server	Zipf parameter (θ)	0.0-1.0
	Data item size	8000 bits
Mobile Transaction	Transaction length (# of operations)	8
	Read operation probability (for update transactions)	0.5
	Read-only to update transactions ratio	0.7
	Mean inter-operation delay	65,536 bit-times (exponentially distributed)
	Mean inter-transaction delay	131,072 bit-times (exponentially distributed)
	The number of mobile transactions	100-1000

4.2 Simulation Results Analysis and Evaluation

The metrics used to measure the performance of the proposed method are the response time, the average number of aborts, and the average number of commits. The performance difference between the proposed method and AOCCRB [7] is shown by varying these parameters' values, and the number of sub-cycles.

4.2.1 The Effect of Sub-cycles on Response Time

The effect of varying the number of sub-cycles, is tested against using a single sub-cycle that indicates the absence of cycle decomposition, on the response time is discussed in this section. As shown in **Fig. 10**, the response time of the proposed AOCCRBSC method is slightly better than that of the AOCCRB [7] when the cycle is not decomposed, and the difference starts to expand as the number of sub-cycles gets larger.

Reasonably, AOCCRBSC has a better response time over AOCCRB even when the cycle is not decomposed because of the usage of the predeclaration, where the cost (time and power) of reading from the cache is much smaller than reading from the channel, in the proposed technique. Moreover, the time spent (and power consumed) by a restarted transaction before abortion is much smaller, in case of using sub-cycles, for the AOCCRBSC method; which leads to a better response time.

Furthermore, the overall response time is enhanced by using AOCCRBSC because of adapting back-off to distribute the aborted update transactions over the next sub-cycles instead of the longer cycles which decreases the time spent by the aborted transaction waiting to restart.

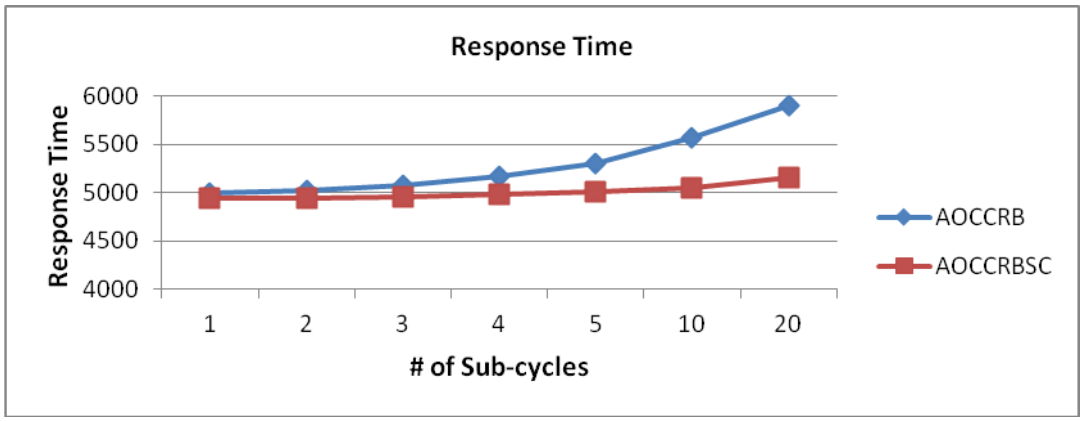

Fig. 10. Response time

4.2.2 The Effect of Sub-Cycles On the Average Number of Aborts

In this section, the average number of locally aborted transactions by the proposed AOCCRBSC method is compared to that of AOCCRB. **Fig. 11** shows that the average number of local aborts by AOCCRBSC is larger than that of AOCCRB. This is because backward validation is more frequently carried out in AOCCRBSC as mobile clients performs the validation at the beginning of each sub-cycle; and in turn conflicts are also more frequently detected, and resolved by locally aborting conflicted transactions.

Moreover, conflicts are detected and resolved by AOCCRBSC for mobile transaction at earlier stages of their execution, which not only increases number of locally aborted mobile transactions but also decreases the cost (in both time and power consumption) of transaction abortion. Obviously, aborting the mobile transaction at the server is much expensive than at mobile clients, the larger number of locally aborted transactions can lead to a better performance. However, read-only transactions are allowed to commit locally whenever they complete their execution before the end of the current sub-cycle, and update transactions may grant commit by the end of the current sub-cycle as well.

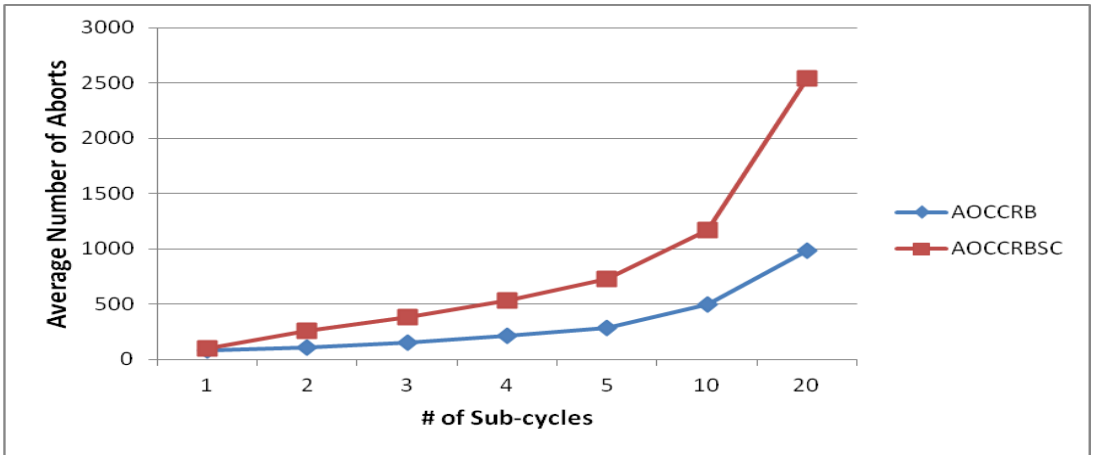

Fig. 11. Average Number of Aborts

4.2.3 The Effect of Sub-Cycles On the Average Number of Commits

In this section, the average number of committed transactions by the proposed AOCCRBSC method is compared to that when AOCCRB is applied, **Fig. 12** shows that the number of committed transactions by AOCCRBSC is the same as for AOCCRB; when the cycle is not decomposed, and starts to get larger as the number of sub-cycles increases.

Only one transaction among all the conflicting update transaction in the cycle, is allowed to commit for both methods when decomposition is not applied, However, AOCCRB does not benefit from cycle decomposition, while AOCCRBSC incrementally accepts transactions from each group of the conflicting update transactions at each number of sub-cycles. In other words, the number of committed update transactions by AOCCRBSC in any cycle is proportional to the number of sub-cycles constituting this cycle, while AOCCRB allows only one conflicted update transaction to commit in each cycle.

Moreover, as the actual reason of conflicts is update transactions; as the rate of update transactions termination increased, the number of conflicts decreases accordingly. In fact, as the probability of (update) transaction to commit is increased, the competition to commit is reduced, which is the case of using AOCCRBSC. Furthermore, when the number of conflict originators decreases, the number of locally committed read-only transactions is increased as well.

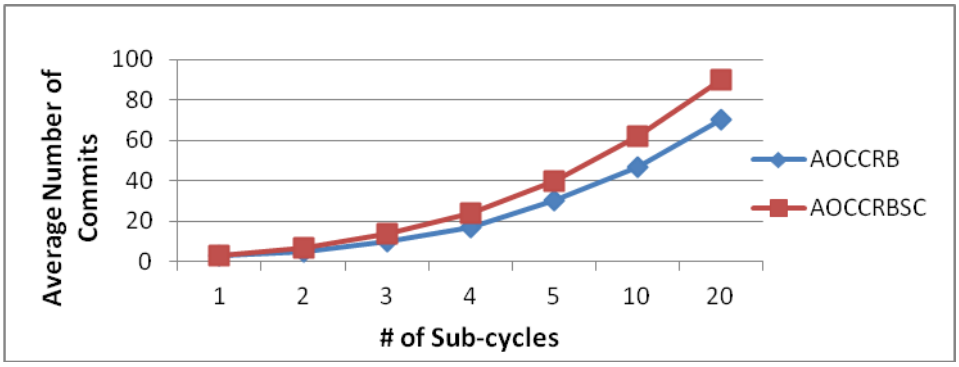

Fig. 12. Average Number of Commits

5. Conclusion and Future Work

Due to the analysis of simulation results in the previous section, AOCCRBSC method has shown a better performance than AOCCRB, and intern is considered as a powerful optimal concurrency control method that suits wireless broadcasting environments.

The enhanced performance shown by AOCCRBSC is represented by gaining better response time, power conservation, and utilized usage of uplink bandwidth over AOCCRB for all predefined simulation setting.

Finally, it is more effective to apply the proposed model, because the response time is still enhanced even in the absence of cycle decomposition because of the predeclaration exploiting, which uses the knowledge obtained from the transaction's failed to execute, allowing the mobile client to read the unprefetched data items from air in the sequence they appear in the cycle instead of that in the rerun transaction.

In the future, the target is to extend the simulator for the testing of STUBcast to improve the performance analysis of the proposed method. Moreover, more evaluation metrics (such

as data currency, uplink and downlink usage …) are to be implemented by the simulator to precisely determine the performance of the proposed method compared to others. Furthermore, the simulation is developed to test more factors (transaction length, cycle size, etc.). Finally, an in depth research to find a mechanism that determines the optimal number of sub-cycles that gives the best performance of the proposed method may be performed.

References

[1] Houling Ji, Victor C.S. Lee, Chi-Yin Chow, Kai Liu, Guoqing Wu, "Coding-based cooperative caching in on-demand data broadcast environments," *Information Sciences*, vol. 385-386, pp. 138-156, 2017. Article (CrossRef Link)

[2] A. Datta, D. E. VanderMeer, A. Celik, and V. Kumar. "Broadcast protocols to support efficient retrieval from databases by mobile users," *ACM Transactions on Database Systems*, vol. 24, no. 1, pp. 1-79, March, 1999. Article (CrossRef Link)

[3] Ali R. Hurson, Sahra Sedigh Sarvestani and Mike Wisely, "Energy-efficient algorithms for data retrieval from indexed parallel broadcast channels," *Sustainable Computing: Informatics and Systems*, vol. 10, pp. 20-35, 2016. Article (CrossRef Link)

[4] J. Juran, A.R. Hurson, N. Vijaykrishnan and S. Kim, "Data Organization and Retrieval on Parallel Air Channels: Performance and Energy Issues," *Wireless Networks*, vol. 10, no. 2, pp. 183–195, 2004. Article (CrossRef Link)

[5] K. Lee, H. Leong and A. Si, "A semantic broadcast scheme for a mobile environment based on dynamic chunking," in *Proc. of 20th IEEE International Conference on Distributed Computing Systems (ICDCS 2000)*, pp. 522-529, 2000. Article (CrossRef Link)

[6] K. Lee, H. Leong and A. Si, "Semantic data access in an asymmetric mobile environment," in *Proc. of Third International Conference on Mobile Data Management (MDM 2002)*, pp. 94-101, 2002. Article (CrossRef Link)

[7] Sunggeun Park and Sungwon Jung, "An energy-efficient mobile transaction processing method using random back-off in wireless broadcast environments," *Journal of Systems and Software*, vol. 82, no. 12, pp. 2012-2022, December, 2009. Article (CrossRef Link)

[8] Sungwon Jung and Keunha Choi, "A concurrency control scheme for mobile transactions in broadcast disk," *Data & Knowledge Engineering*, vol. 68, no. 10, pp. 926-945, October, 2009. Article (CrossRef Link)

[9] Young-Kyoon Suh, Richard T. Snodgrass and Sabah Currim, "An empirical study of transaction throughput thrashing across multiple relational DBMSes," *Information Systems*, vol. 66, pp. 119-136, June, 2017. Article (CrossRef Link)

[10] Md. Anisur Rahma, "An Efficient Concurrency Control Technique for Mobile Database Environment," *Global Journal of Computer Science and Technology Software & Data Engineering*, vol. 13, no. 2, 2013. Article (CrossRef Link)

[11] Qasim Abbas, Hammad Shafiq, Imran Ahmad and Sridevi Tharanidharan, "Concurrency control in distributed database system," in *Proc. of 2016 International Conference on Computer Communication and Informatics (ICCCI)*, 2016. Article (CrossRef Link)

[12] Jan L. Harrington, "Chapter 22 - Concurrency Control," *Relational Database Design and Implementation*, 4th edition, pp. 449-470, Morgan Kaufmann, Boston, 2016. Article (CrossRef Link)

[13] O. A. Rawashdeh, H. A. Muhareb and N. A. Al-Sayid, "An optimistic approach in distributed database concurrency control," in *Proc. of 2013 5th International Conference on Computer Science and Information Technology*, pp. 71-75, 2013. Article (CrossRef Link)

[14] Q. Zheng, K. Zheng, H. Zhang and V. C. M. Leung, "Delay-Optimal Virtualized Radio Resource Scheduling in Software-Defined Vehicular Networks via Stochastic Learning," *IEEE Transactions on Vehicular Technology*, vol. 65, no. 10, pp. 7857-7867, Octpber, 2016. Article (CrossRef Link)

[15] Muhammad Baqer Mollah, Md. Abul Kalam Azad and Athanasios Vasilakos, "Security and privacy challenges in mobile cloud computing: Survey and way ahead," *Journal of Network and Computer Applications*, vol. 84, no. 15, pp. 38-54, 2017. Article (CrossRef Link)

[16] Il Young Chung, B. Bhargava, M. Mahoui and L. Lilien, "Autonomous transaction processing using data dependency in mobile environments," in *Proc. of The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'03),* pp. 138-144, 2003. Article (CrossRef Link)

[17] SangKeun Lee, Chong-Sun Hwang and M. Kitsuregawa, "Using predeclaration for efficient read-only transaction processing in wireless data broadcast," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 6, pp. 1579-1583, Nov.-Dec. 2003. Article (CrossRef Link)

[18] SangKeun Lee, Chong-Sun Hwang and M. Kitsuregawa, "Efficient, Energy Conserving Transaction Processing in Wireless Data Broadcast," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 9, pp. 1225-1238, Sept. 2006. Article (CrossRef Link)

[19] E. Pitoura and P. K. Chrysanthis, "Multiversion data broadcast," *IEEE Transactions on Computers*, vol. 51, no. 10, pp. 1224-1230, October, 2002. Article (CrossRef Link)

[20] Xiangdong Lei, Yuelong Zhao, Songqiao Chen and Xiaoli Yuan, "Concurrency control in mobile distributed real-time database systems," *Journal of Parallel and Distributed Computing*, vol. 69, no. 10, pp. 866-876, 2009. Article (CrossRef Link)

[21] Yan Huang and Yann-Hang Lee, "STUBcast - efficient support for concurrency control in broadcast-based asymmetric communication environment,"in *Proc. of Proceedings Tenth International Conference on Computer Communications and Networks (Cat. No.01EX495)*, pp. 262-267, 2001. Article (CrossRef Link)

Ahmad Al-Qerem Graduated in applied mathematics and MSc in Computer Science at the Jordan University of Science and Technology in 1997 and 2002, respectively. After that, he was appointed as full-time lecturer at the Zarqa University and also a part-time lecturer at the Arab Open University. He has also held a post in the Ministry of Labor. He obtained a PhD from Loughborough University, UK. His research interests are in performance and analytical modeling, mobile computing environments, protocol engineering, communication networks, transition to IPv6, and transaction processing. He has published several papers in various areas of computer science. Currently, he has a full academic post as associate professor and the head of the Department of Internet Technology at Zarqa University - Jordan.
Email: ahmad_qerm@zu.edu.jo

Ala Hamarsheh is an assistant professor at the Faculty of Engineering and Information Technology of the Arab American University - Jenin. He obtained a PhD in engineering sciences from Vrije Universiteit Brussel (VUB)/Brussels-Belgium in 2012. He graduated in computer science at the Faculty of Science, Birzeit University, Palestine, in 2000. He obtained an MSc degree in computer science at the Kind Abdullah II School for IT, The University of Jordan, Jordan, in 2003. He has published numerous papers in international refereed journals and conferences.
E-mail: ala.hamarsheh@aauj.edu

Yaser A. Al-Lahham received the B.S degree from University of Jordan in 1985, the M.S. degree from Arab Academy (Jordan) in 2004, and the PhD in Computer science from Bradford University (UK) in 2009. He is working as an assistant professor in the Department of Computer Science at Zarqa University in Jordan. His research interest includes P2P information retrieval systems, text clustering, and Databases.
Email: yasirlhm@zu.edu.jo

Mujahed Eleyat is currently an Assistant Professor in the Department of Computer Systems Engineering in the Faculty of Engineering and Information Technology at the Arab American University. He received his B.A. in Electrical Engineering from Birzeit University and his Master and Ph.D degrees from University of Arkansas and Norwegian University of Science and Technology respectively. Dr. Eleyat areas of expertise include high performance computing, embedded systems, computer architecture, and mobile computing.
Email: mujahed.eleyat@aauj.edu