We are IntechOpen,
the world's leading publisher of
Open Access books
Built by scientists, for scientists

6,900
Open access books available

186,000
International authors and editors

200M
Downloads

154
Countries delivered to

Our authors are among the
TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS
BOOK CITATION INDEX
INDEXED

WEB OF SCIENCE™
Selection of our books indexed in the Book Citation Index
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

A Low-Overhead Non-Block Check Pointing and Recovery Approach for Mobile Computing Environment

Bidyut Gupta[1], Ziping Liu[2] and Sindoora Koneru[1]
[1]*Computer Science Department, Southern Illinois University, Carbondale,*
[2]*Computer Science Department, Southeast Missouri State University, Cape Girardeau,*
USA

1. Introduction

Check pointing/rollback-recovery strategy is used for providing fault-tolerance to distributed applications (Y.M. Wang, 1997; M. Singhal & N. G. Shivaratri, 1994; R. E. Strom and S. Yemini 1985; R. Koo & S. Toueg, 1987; S. Venkatesan et al.,1997; G. Cao & M. Singhal, 1998; D. Manivannan & M. Singhal, 1999). A checkpoint is a snapshot of the local state of a process, saved on local nonvolatile storage to survive process failures. A global checkpoint of an n-process distributed system consists of n checkpoints (local) such that each of these n checkpoints corresponds uniquely to one of the n processes. A global checkpoint M is defined as a consistent global checkpoint if no message is sent after a checkpoint of M and received before another checkpoint of M (Y.M. Wang, 1997). The checkpoints belonging to a consistent global checkpoint are called globally consistent checkpoints (GCCs). The set of such checkpoints is also known as recovery line.

There are two fundamental approaches for checkpointing and recovery. One is the asynchronous approach and the other one is the synchronous approach (D. K. Pradhan and N. H. Vaidya 1994; R. Baldoni et al. 1999; R. Koo & S. Toueg, 1987; S. Venkatesan et al.,1997; G. Cao & M. Singhal, 1998; D. Manivannan & M. Singhal, 1999).

Synchronous approach assumes that a single process invokes the algorithm periodically to take checkpoints. This process is known as the initiator process. This scheme is termed as synchronous since the processes involved coordinate their local check pointing actions such that the set of all recent checkpoints in the system is guaranteed to be consistent. The scheme assumes that no site involved in the distributed computing fails during the execution of the check pointing scheme. In its most fundamental form, it works in two phases as follows.

In the first phase the initiator process takes a tentative checkpoint and requests all other processes to take their respective tentative checkpoints. Each process informs the initiator process that it has taken it.

In the second phase, after receiving such information from all processes the initiator process asks each process to convert its tentative checkpoint to a permanent one. That is, each process saves its checkpoint in nonvolatile storage. During the execution of the scheme each process suspends its underlying computation related to an application. Thus, each process remains blocked each time the algorithm is executed. The set of checkpoints so taken are globally consistent checkpoints, because there is no orphan message with respect to any two checkpoints. It may be noted that a message is an orphan with respect to the recent checkpoints of two processes if its receiving event is recorded in the recent checkpoint of the receiver of the message, but its sending event is not recorded in the recent checkpoint of its sender.

This synchronous approach has the following two major drawbacks. First, The coordination among processes while taking checkpoints is actually achieved through the exchange of additional (control) messages, for example the requests from the initiator and the replies to the initiator. It causes some delay (known as synchronization delay) during normal operation. The second drawback is that processes remain blocked during check pointing. It contributes significantly to the amount of delay during normal operation. However, the main advantage is that the set of the checkpoints taken periodically by the different processes always represents a consistent global checkpoint. So, after the system recovers from a failure, each process knows where to rollback for restarting its computation again. In fact, the restarting state will always be the most recent consistent global checkpoint. Therefore, recovery is very simple. On the other hand, if failures rarely occur between successive checkpoints, then the synchronous approach places unnecessary burden on the system in the form of additional messages and delay. Hence, compared to the asynchronous approach, taking checkpoints is more complex while recovery is much simpler. Observe that synchronous approach is free from any domino effect (B. Randell, 1979).

In the asynchronous approach, processes take their checkpoints independently. So, taking checkpoints is very simple as there is no coordination needed among processes while taking checkpoints. Obviously, there is no blocking of the processes while taking checkpoints unlike in the synchronous approach. After a failure occurs, a procedure for rollback-recovery attempts to build a consistent global checkpoint. However, in this approach because of the absence of any coordination among the processes there may not exist a recent consistent global checkpoint which may cause a rollback of the computation. This is known as domino effect (B. Randell, 1975; K. Venkatesh et al., 1987). Observe that the cause for domino effect is the existence of orphan messages. In the worst case of the domino effect, after the system recovers from a failure all processes may have to roll back to their respective initial states to restart their computation again. In general, to minimize the amount of computation undone during a rollback, all messages need to be saved (logged) at each process.

Besides these two fundamental approaches there is another approach known as communication induced check pointing approach (J. Tsai et al., 1998; R. Baldoni et al., 1997; J. M. Helary et al., 2000). In this approach processes coordinate to take checkpoints via piggybacking some control information on application messages. However this coordination does not guarantee that a recent global checkpoint will be consistent. This means that this approach also suffers from the domino effect. Therefore, a recovery algorithm has to search for a consistent global checkpoint before the processes can restart their computation after recovery from a failure. In this approach taking checkpoints is simpler than synchronous approach while the recovery process is more complex.

B. Gupta et al., 2002 have proposed a simple and fast roll-forward check pointing scheme that can also be used in distributed mobile computing environment. The direct-dependency concept used in the communication-induced check pointing scheme has been applied to basic checkpoints (the ones taken asynchronously) to design a simple algorithm to find a consistent global checkpoint. Both blocking and non-blocking schemes have been proposed. In the blocking approach direct-dependency concept is implemented without piggybacking any extra information with the application messages. The use of the concept of forced checkpoints ensures a small re-execution time after recovery from a failure. The proposed approach offers the main advantages of both the synchronous and the asynchronous approaches, i.e. simple recovery and simple way to create checkpoints. Besides, the algorithm produces reduced number of checkpoints. To achieve these, the algorithm uses very simple data structure per process, that is, each process maintains only a Boolean flag and an integer variable. Since each process independently takes its decision whether to take a forced checkpoint or not, it makes the algorithm simple, fast, and efficient. The advantages stated above also ensure that the algorithm can work efficiently in mobile computing environment.

There also exist some other efficient non-blocking algorithms (G. Cao & M. Singhal, 2001; E. N. Elnozahy et al., 1992; L. M. Silva & J. G. Silva, 1992); however they require significant number of control (system) messages to determine a consistent global checkpoint of the system. In (G. Cao & M. Singhal, 2001), the authors have proposed an efficient non-blocking coordinated check pointing scheme that offers minimum number of check points. They have introduced the concept of mutable checkpoint which is neither a tentative checkpoint nor a permanent checkpoint to design their check pointing scheme for mobile computing environment. Mutable checkpoints can be saved either in the main memory or local disks. It has been shown that the idea of mutable checkpoints helps in the efficient utilization of wireless bandwidth of the mobile environment. In general, it may be stated that the ideas of non-blocking check pointing, reduction in the number of checkpoints to be taken, and using less number of system messages may offer significant advantage particularly in case of mobile computing, because it helps in the efficient use of the limited resources of mobile computing environment, viz. limited wireless bandwidth, and mobile hosts' limited battery power and memory.

In this context, note that after recovery from a failure even if the processes restart from their respective checkpoints belonging to a recovery line, still it not necessarily ensures correctness of computation. To achieve it, any application message that may become a lost message because of the failure must be identified and resent to the appropriate receiving process. The responsibility of the receiving process is that it must execute all such lost messages following the order of their arrival before the occurrence of the failure (D. B. Johnson & W. Zwaenepoel, 1987; M. L. Powell & D. L. Presotto, 1983; L. Alvisi & K. Marzullo, 1995).

An example of a lost message is shown in Fig. 1. In this figure, after the system recovers from the failure f, if the two processes P_i and P_j restart from their respective checkpoints C_i and C_j, then message m will be treated as a lost message. The reason is that process P_j does not have a record of the receiving event of the message m whereas process P_i has the record of sending it in its checkpoint C_i. Therefore when the processes restart, P_i will not send message m again to P_j since it knows that it already sent it once. However this will lead to wrong computation, because P_j aftet its roll back to its last checkpoint needs the message m for its computation. In such a situation, for correct computation this lost message m has to be identified and process P_i must resend it to process P_j after the system restarts.

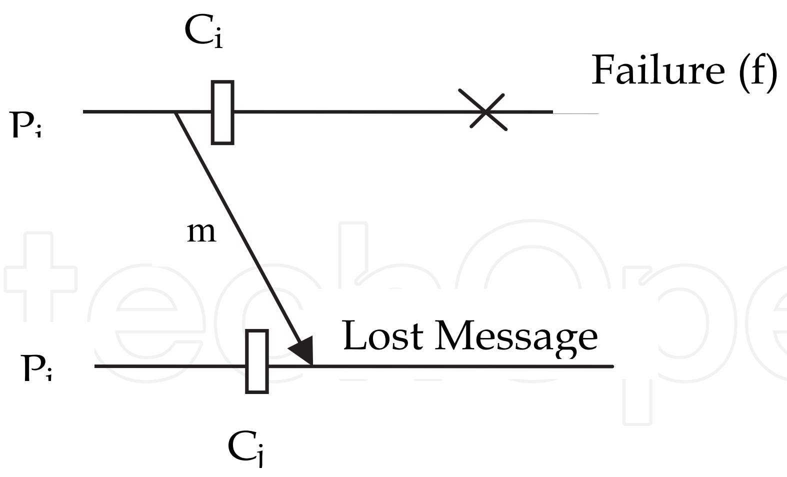


Fig. 1. Message m is a lost message

The objective of this work, is to design a check pointing and recovery scheme for distributed computing environment which is also very suitable for distributed mobile computing environment. It considers both determination of a recovery line and resending of all lost messages to ensure correctness of computation. First, a fast recovery algorithm is presented that determines a recovery line that guarantees the absence of any orphan message with respect to the checkpoints belonging to the recovery line. Then the existing idea on sender-based message logging approach for distributed computing (D. B. Johnson & W. Zwaenepoel, 1987) is applied to identify and resend any lost messages. It helps a receiving process, after it restarts, to process these messages following the order of their arrival before the occurrence of the failure. Thus taking into account both orphan messages and lost messages will ensure correctness of computation.

The presented check pointing algorithm is a non-blocking synchronous algorithm. It means application processes are not suspended during check pointing. The non-blocking algorithm does not require that all processes take their checkpoints; rather only those processes that have sent some message(s) after their last check points will take checkpoints during check pointing. It is shown that this algorithm outperforms the one in (G. Cao & M. Singhal, 2001) mainly from the viewpoint of using much less number of system (control) messages. As pointed out earlier that non-blocking check pointing along with the reduced number of checkpoints to be taken and less number of system messages may offer significant advantage particularly in case of mobile computing, because it helps in the efficient use of the limited resources of mobile computing environment, viz. limited wireless bandwidth, and mobile hosts' limited battery power and memory.

This work is organized as follows: in Sections 2 and 3 we have stated the system model and the necessary data structures respectively. In Section 4, using an example we have explained the main idea about when a process needs to take a checkpoint by using some very simple data structures. We have stated some simple observations necessary to design the algorithm. In Section 5 we have presented the non blocking check pointing algorithm along with its performance, and presented a scheme for handling lost messages. In Section 6 we have discussed its suitability for mobile computing systems. Section 7 draws the conclusions.

2. System model

The distributed system has the following characteristics (R. Koo & S. Toueg, 1987; S. Venkatesan et al.,1997; P. Jalote, 1998): Processes do not share memory and communicate via messages sent through channels Channels can lose messages. However, they are made virtually lossless and order of the messages is preserved by some end-to-end transmission protocol. Message sequence numbers may be used to preserve the order. When a process fails, all other processes are notified of the failure in finite time. We also assume that no further processor (process) failures occur during the execution of the algorithm. In fact, the algorithm may be restarted if there are further failures. Processes are piecewise deterministic in the sense that from the same state, if given the same inputs, a process executes the same sequence of instructions.

3. Data structures

Let us consider a set of n processes, $\{P_0, P_1, \ldots, P_{n-1}\}$ involved in the execution of a distributed algorithm. Each process P_i maintains a Boolean flag c_i. The flag is initially set at zero. It is set at 1 only when process P_i sends its first application message after its latest checkpoint. It is reset to 0 again when process P_i takes a checkpoint. Flag c_i is stored in local RAM of the processor running process P_i. A message sent by P_i will be denoted as m_i.

As in the classical synchronous approach (M. Singhal & N. G. Shivaratri, 1994), we assume that an initiator process initiates the check pointing algorithm. It helps the n processes to take their individual checkpoints synchronously, i.e. the checkpoints taken will be globally consistent checkpoints. We further assume that any process in the system can initiate the check pointing algorithm. This can be done in a round-robin way among the processes. To implement it, each process P_i maintains a variable CLK_i initialized at 0. It also maintains a variable, $counter_i$ which is initially set to 0 and is incremented by 1 each time process P_i initiates the algorithm. In addition, process P_i maintains an integer variable N_i which is initially set at 0 and is incremented by 1 each time the algorithm is invoked. Note the difference between the variables $counter_i$ and N_i. A control (request) message M_c is broadcasted by a process initiating the check pointing algorithm to the other (n-1) processes asking them to take checkpoints if necessary.

In the next section, we explain with an illustration the idea we have applied to reduce the number of checkpoints to be created in the non blocking synchronous check pointing scheme proposed in the work.

4. An illustration

In synchronous check pointing scheme, all involved processes take checkpoints periodically which are mutually consistent. However, in reality, not all the processes may need to take checkpoints to determine a set of the GCCs.

The main objective of this work is to design a simple scheme that helps the n processes to decide easily and independently whether to take a checkpoint when the check pointing algorithm is invoked. If a process decides that it does not need to take a checkpoint, it can resume its computation immediately. This results in faster execution of the distributed algorithm. Below we illustrate with an example how a process decides whether to take a checkpoint or not.

Consider the following scenario of a distributed system of two processes P_i and P_j only. It is shown in Fig. 2. Assume that their initial checkpoints are C_i^0 and C_j^0 respectively. According to the synchronous approach, P_i and P_j have to take checkpoints periodically. Suppose that the time period is T. Before time T, P_i has sent an application message m_1 to P_j. Now at time T, an initiator process sends the message M_c asking both P_i and P_j to take their checkpoints, which must have to be consistent.

Process P_i checks its flag and finds that $c_i = 1$. Therefore P_i decides to take its checkpoint C_i^1. Thus because of the presence of C_i^1, message m_1 can never be an orphan. Also, at the same time P_j checks if its flag $c_j = 1$. Since it is not, therefore process P_j decides that it does not need to take any checkpoint. The reason is obvious. This illustrates the basic idea about how to reduce the number of checkpoints to be taken. Now we observe that checkpoints C_j^0 and C_i^1 are mutually consistent.

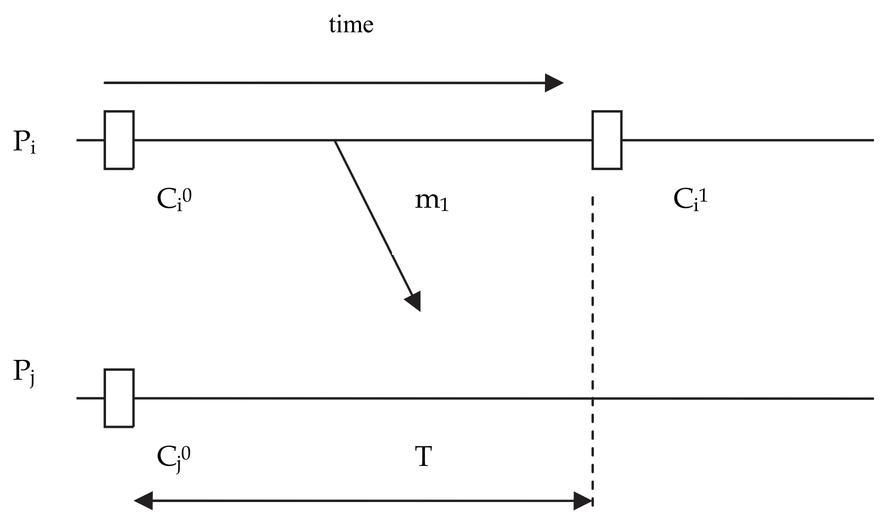


Fig. 2. C_i^1 and C_j^0 are mutually consistent

The above discussion shows the simplicity involved in taking a decision about whether to take a checkpoint or not. Note that the decision taken by a process P_j whether it needs to take a checkpoint is independent of the similar decision taken by the other process. It may be noted that keeping a copy of each of the flags c_i and c_j in the respective local RAMs of the processors running P_i and P_j can save some time as it is more time consuming to fetch them if they are stored in stable storage than to fetch them from the respective local RAMs.

Below we state some simple but important observations used in the proposed algorithm.

Theorem1: Consider a system of n processes. If $c_j = 1$, where C_j^k is the latest checkpoint of process P_j , then some message(s) sent by P_j to other processes may become orphan.

Proof: The flag c_j is reset to 0 at every checkpoint. It can have the value 1 only between two successive checkpoints of any process P_j if and only if process P_j sends at least one message m between the checkpoints. Therefore, $c_j = 1$ means that P_j is yet to take its next checkpoint following C_j^k. Therefore, the message (s) sent by P_j after its latest checkpoint C_j^k are not yet recorded. Now if some process P_m receives one or more of these messages sent by P_j and then takes its latest checkpoint before process P_j takes its next checkpoint C_j^{k+1}, then these received messages will become orphan. Hence the proof follows.

Theorem 2: If at any given time t, $c_j = 0$ for process P_j with C_j^{k+1} being its latest checkpoint, then none of the messages sent by P_j remains an orphan at time t.

Proof: Flag c_j can have the value 1 between two successive checkpoints, say C_j^k and C_j^{k+1}, of a process P_j if and only if process P_j has sent at least one message m between these two checkpoints. It can also be 1 if P_j has sent at least a message after taking its latest checkpoint. It is reset to 0 at each checkpoint. On the other hand, it will have the value 0 either between two successive checkpoints, say C_j^k and C_j^{k+1}, if process P_j has not sent any message between these checkpoints, or P_j has not sent any message after its latest checkpoint. Therefore, $c_j = 0$ at time t means either of the following two: (i) $c_j = 0$ at C_j^{k+1} and this checkpoint has been taken at time t. It means that any message m sent by P_j (if any) to any other process P_m between C_j^k and C_j^{k+1} must have been recorded by the sending process P_j at the checkpoint C_j^{k+1}. So the message m can not be an orphan. (ii) $c_j = 0$ at time t and P_j has taken its latest checkpoint C_j^{k+1} before time t. It means that process P_j has not sent any message after its latest checkpoint C_j^{k+1} till time t. Hence at time t there does not exist any orphan message sent by P_j after its latest checkpoint.

5. Problems associated with non-blocking approach

We explain first the problems associated with non-blocking approach. After that we will state a solution. The following discussion although considers only two processes, still the arguments given are valid for any number of processes. Consider a system of two processes P_i and P_j as shown in Fig. 3. Assume that the check pointing algorithm has been initiated by process P_i and it has sent the request message M_c to P_j asking it to take a checkpoint if necessary. As pointed earlier that both processes will act independently, therefore P_i takes its checkpoint C_i^1 because its flag $c_i = 1$. Let us assume that P_i now immediately sends an application message m_i to P_j. Suppose at time (T + €), where € is very small with respect to T, P_j receives m_i. Still P_j has not received M_c from the initiator process. So, P_j processes the message. Now the request message M_c from P_i arrives at P_j. Process P_j finds that its $c_j = 1$. So it decides to take a checkpoint C_j^1. We find that message m_i has become an orphan due to the checkpoint C_j^1. Hence, $C_i^1$and C_j^1 cannot be consistent.

5.1 Solution

To solve this problem, we propose that a process be allowed to send both piggybacked and non –piggybacked application messages. We explain the idea below.

Each process P_i maintains an integer variable N_i, initially set at 0 and is incremented by 1 each time process P_i receives the request message M_c from the initiator process. In the event that process P_i itself is the initiator, then also it increments N_i by 1 immediately after the initiation of the algorithm. That is, the variable N_i represents how many times the check pointing algorithm has been executed including the current one (according to the knowledge of the process P_i). Note that at any given time t, for any two processes P_i and P_j, their corresponding variables N_i and N_j may not have the same values. It depends on which process has received the request message M_c first. However it is obvious that $| N_i - N_j |$ is either 0 or 1.

Below we state the solution for a two process system. The idea used in this solution is similarly applicable for an n process system as well.

Consider a distributed system of two processes P_i and P_j only. Without any loss of generality assume that P_i initiates the algorithm by sending the message M_c to process P_j and it is the the kth execution of the algorithm, that is, $N_i = k$. We also assume that process P_i now has taken its decision whether to take a checkpoint or not, and then has taken appropriate action to implement its decision. Suppose P_i now wants to send an application message m_i for the first time to P_j after it has finished participating in the kth execution of the check pointing algorithm. Observe that P_i has no idea whether P_j has received the message M_c corresponding to this kth execution of the algorithm and has already implemented its check pointing decision or not. To make sure that the message m_i can never be an orphan, P_i piggybacks m_i with the variable N_i. Process P_j receives the piggybacked message $<m_i , N_i >$ from P_i. We now explain below why the message m_i can never been an orphan. Note that $N_i = k$; i.e. it is the kth execution of the algorithm that process P_i has last been involved with. It means the following to the receiver P_j of this message:

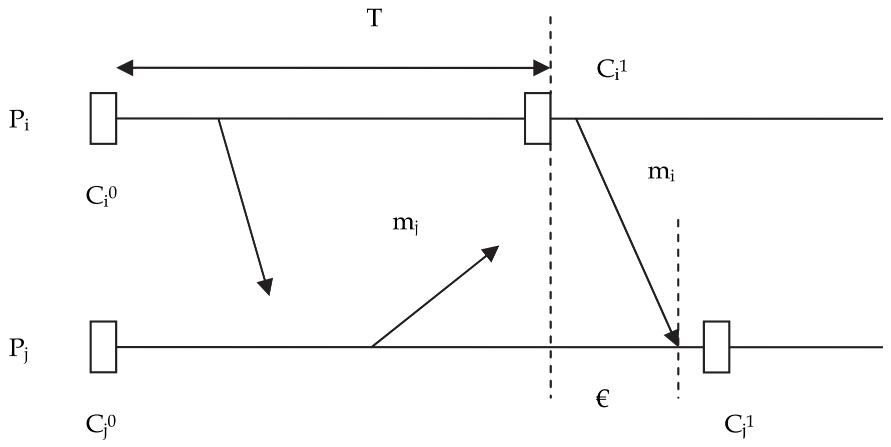


Fig. 3. C_i^1 and C_j^1 are not mutually consistent

1. process P_i has already received M_c from the initiator process for the kth execution of the algorithm,
2. P_i has taken a decision whether to take a checkpoint or not and has taken appropriate action to implement its decision,
3. P_i has resumed its normal operation and then has sent this piggybacked application message m_i.
4. the sending event of message m_i has not yet been recorded by P_i.

Since the message contains the variable N_i, process P_j compares N_i and N_j to determine if it has to wait to receive the request message M_c. Based on the results of the comparison process P_j takes one of the following three actions (so that no message received by it is an orphan), as stated below in the form of the following three observations:

Observation 1: If N_i (= k) > N_j (= k-1), process P_j now knows that the kth execution of the check pointing algorithm has already begun and so very soon it will also receive the message M_c from the initiator process associated with this execution. So instead of waiting for M_c to arrive, it decides if it needs to take a checkpoint and implements its decision, and then processes the message m_i. After a little while when it receives the message M_c it just ignores it. Therefore, message m_i can never be an orphan.

Observation 2: If $N_i = N_j = k$, like process P_i, process P_j also has received already the message M_c associated with the latest execution (kth) of the check pointing algorithm and has taken its check pointing decision and has already implemented that decision. Therefore, process P_j now processes the message m_i. It ensures that message m_i can never be an orphan, because both the sending and the receiving events of message m_i have not been recorded by the sender P_i and the receiver P_j respectively.

Observation 3: Process P_i does no more need to piggyback any application message to P_j till the (k+1)th invocation (next) of the algorithm. The reason is that after receiving the piggybacked message $<m_i, N_i>$, P_j has already implemented its decision whether to take a checkpoint or not before processing the message m_i. If it has taken a checkpoint, then all messages it receives from P_i starting with the message m_i can not be orphan. So it processes the received messages. Also if P_j did not need to take a checkpoint during the kth execution of the algorithm, then obviously the messages sent by P_i to P_j staring with the message m_i till the next invocation of the algorithm can not be orphan. So it processes the messages.

Therefore, for an n process distributed system, a process P_i piggybacks only its first application message sent (after it has implemented its check pointing decision for the current execution of the algorithm and before its next participation in the algorithm) to a process P_j, where $j \neq i$, and $0 \leq j \leq n-1$.

5.2 Algorithm non-blocking

Below we describe the algorithm. It is a single phase algorithm since an initiator process interacts with the other processes only once via the control message M_c.

```
At each process $P_i (1 \leq i \leq n)$
  if $CLK_i = (i + (counter_i * n)) * T$  //when its turn to initiate the checkpointing procedure
    $counter_i = counter_i + 1$;
    $N_i = N_i + 1$;
    broadcasts $M_c$ to $(n-1)$ other processes;

    if $c_i = 1$ //at least one message it has sent after its last checkpoint
      takes checkpoint$C_i$;
      $c_i = 0$;
      continues its normal operation;

    else //if it decides not to take a checkpoint
      continues its normal operation;
else if $P_i$ receives $M_c$
    $N_i = N_i + 1$;
    if $c_i = 1$ //at least one message it has sent after its last checkpoint
      takes checkpoint$C_i$;
      $c_i = 0$;
      continues its normal operation;
    else
      continues its normal operation;
```

```
else if  $P_i$ receives a piggybackedmessage $< m_j, N_j >$ && $P_i$ has not yet received $M_c$ for the
        current execution of the check pointing procedure

    $N_i = N_i + 1$;

    if $c_i = 1$ //at least one message it has sent after its last checkpoint
        $c_i = 0$;
        takes checkpoint $C_i$ without waiting for $M_c$;
        processes the received message $m_j$;
        continues its normal operation and ignores $M_c$, when received for the
        current execution of the checkpointing procedure;

    else
        processes any received message $m_j$;
        continues its normal operation and ignores $M_c$, when received for the
        current execution of the check pointing procedure;
else
        continues its normal operation;
```

Proof of Correctness : In the first 'if else' and 'else if' blocks of the pseudo code, each process P_i decides based on the value of its flag c_i whether it needs to take a checkpoint. If it has to take a checkpoint, it resets c_i to 0. Therefore, in other words, each process P_i makes sure using the logic of Theorem 2 that none of the messages, if any, it has sent since its last checkpoint can be an orphan. On the other hand, if P_i does not take a checkpoint, it means that it has not sent any message since its previous checkpoint.

In the second 'else if' block each process P_i follows the logic of Observations 1, 2, and 3, which ever is appropriate for a particular situation so that any application message (piggybacked or not) received by P_i before it receives the request message M_c can not be an orphan. Besides none of its sent messages, if any, since its last checkpoint can be an orphan as well (following the logic of Theorems 1 and 2).

Since Theorem 2, and Observations 1, 2, and 3 guarantee that no sent or received message by any process P_i since its previous checkpoint can be an orphan and since it is true for all participating processes, therefore, the algorithm guarantees that the latest checkpoints taken during the current execution of the algorithm and the previous checkpoints (if any) of those processes that did not need to take checkpoints during the current execution of the algorithm are globally consistent checkpoints.

5.3 Performance

We use the following notations (and some of the analysis from (G. Cao & M. Singhal, 2001) to compare our algorithm with some of the most notable algorithms in this area of research, namely (R. Koo & S. Toueg, 1987; G. Cao & M. Singhal, 2001; E. N. Elnozahy et al., 1992). The analytical comparison is given in Table 1.

In this Table:

C_{air} is cost of sending a message from one process to another process;
C_{broad} is cost of broadcasting a message to all processes;
n_{min} is the number of processes that need to take checkpoints.
n is the total number of processes in the system;
n_{dep} is the average number of processes on which a process depends;
T_{ch} is the check pointing time;

Algorithm	**Blocking time**	**Messages**	**Distributed**
Koo-Toueg [3]	$n_{min} * T_{ch}$	$3 * n_{min} * n_{dep} * C_{air}$	Yes
Elnozahy [8]	0	$2 * C_{broad} + n * C_{air}$	No
Cao-Singhal [7]	0	$\approx 2 * n_{min} * C_{air} + \min(n_{min} * C_{air}, C_{broad})$	Yes
Our Algorithm	0	C_{broad}	Yes

Table 1. System Performance

Figs. 4 and 5 illustrate how the number of control messages (system messages) sent and received by processes is affected by the increase in the number of the processes in the system.

In Fig. 4, n_{dep} factor is considered being 5% of the total number of processes in the system and C_{broad} is equal to C_{air} (assuming that special hardware is used to facilitate broadcasting – which is not the case most of the times). As Fig. 4 shows, the number of messages does not increase with the increase of the number of the processes in our approach unlike other approaches.

In Fig. 5 we have considered absence of any special hardware for broadcasting and therefore assumed C_{broad} to be equal to $n * C_{air}$. In this case, although the number of messages does increase in our approach, but it stays smaller compared to other approaches when the number of the processes is higher than 7 (which is the case most of the time).

5.4 Handling of lost messages

The sender-based message logging scheme proposed for distributed computing (D. B. Johnson & W. Zwaenepoel, 1987) to identify and resend lost messages is used in this work. This scheme has been the choice since it does not require message ordering, and message logging is done asynchronously. We apply it in the following way.

When a sending process, say P_i sends a message m to a process P_k, the message m is piggybacked with a send sequence number (SSN) which represents the number of messages sent by this process. The sender also logs the message m and its SSN in its local log. The receiving process P_k will assign a receive sequence number (RSN) to the message m, which represents the number of messages received by P_k. The RSN is incremented each time P_k receives a message. It then sends the RSN back to the sender P_i. After receiving the RSN corresponding to m, the sender records the RSN with the log of the message m. Thus message m is called a fully logged message. This local log is saved in stable storage when P_i takes its next checkpoint. Process P_i then sends an acknowledgement, ack to the receiver. In the meantime after sending the RSN to P_i, process P_k continues its execution, but cannot send any message until it has received the ack. Note that if the receiver fails before sending the RSN of the message m, the log of m does not have the RSN. In such a situation message m is called partially logged.

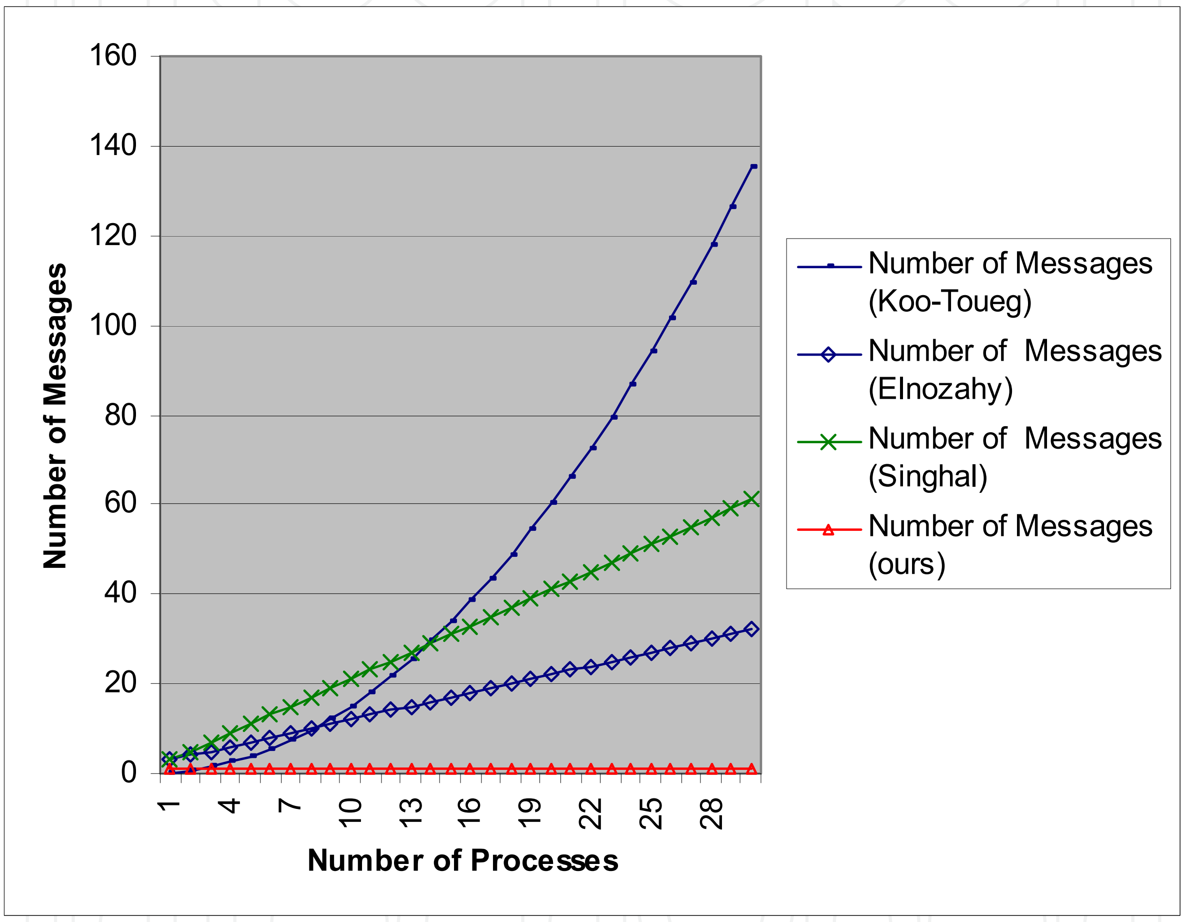


Fig. 4. Number of messages vs. number of processes for four different approaches when $C_{broad} = C_{air}$

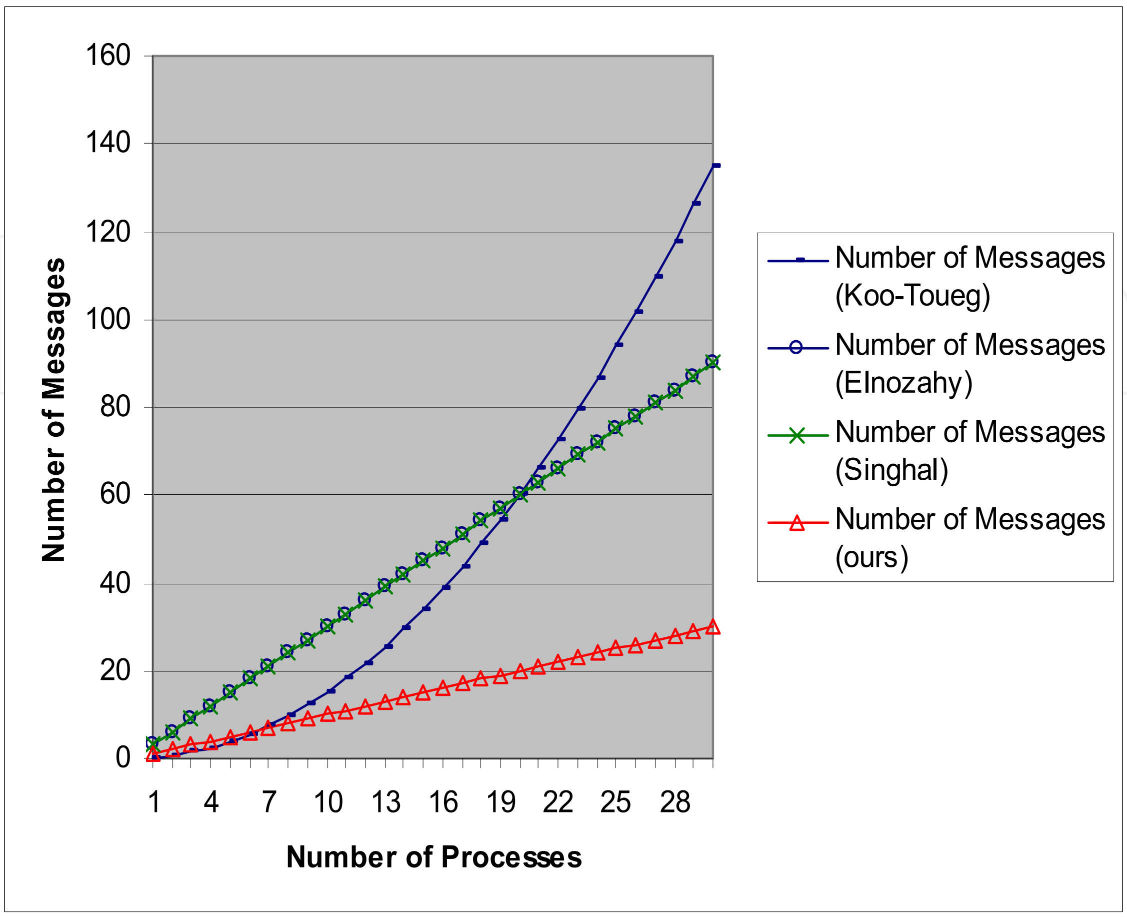


Fig. 5. Number of messages vs. number of processes for four different approaches when $C_{broad} = n * C_{air}$

Recovery is performed when P_k fails. It restarts from its checkpoint that belongs to the recovery line as determined by the check pointing algorithm. Now P_k looks for those (if any) messages such that their sending events have already been recorded in the respective senders' checkpoints and the receiving events have not been recorded in P_k's checkpoint belonging to the recovery line. These are the lost messages. To get back these messages, the receiver broadcasts a request to all processes for resending the lost messages. At this time the receiver also sends the value of the SSNs for different sender processes. Every sender then resends only those messages with a higher SSN that were sent to P_k before the failure.

The messages received by P_k from the senders are consumed by P_k in the order of their RSNs. Since the messages that were assigned an RSN by the receiver form a total order, therefore process P_k gets the same sequence of messages as it did before the failure and therefore executes the same sequence of instructions as it did before the failure.

Next, P_k receives the partially logged messages following the fully logged ones. These partially logged messages do not have any RSN values attached to them. So there is no total ordering imposed on them by P_k. However, according to the work in (D. B. Johnson & W. Zwaenepoel, 1987) the receiver was constrained from communicating with any process if the ack for any message it received is pending. Therefore any order in which these partially logged messages are resent to P_k is acceptable (D. B. Johnson & W. Zwaenepoel, 1987).

Observe that in our approach there does not exist any orphan message between any two checkpoints belonging to the recovery line. Hence the mechanism to handle orphan messages in (D. B. Johnson & W. Zwaenepoel, 1987) is not needed in our approach. Thus absence of any orphan and lost messages ensures correctness of computation.

6. Suitability for mobile computing environment

Consider a distributed mobile computing environment. In such an environment, only limited wireless bandwidth is available for communication among the computing processes. Besides, the mobile hosts (MH) have limited battery power and limited memory. Therefore, it is required that, any distributed application *P* running in such an environment must make efficient use of the limited wireless bandwidth, and mobile hosts' limited battery power and memory. Below we show that the proposed algorithm satisfies all the above three requirements.

1. The first requirement about the efficient use of the bandwidth is satisfied by our check pointing algorithm, because the presented algorithm is a single phase algorithm unlike any other existing algorithms (R. Koo & S. Toueg, 1987; D. Manivannan & M. Singhal, 1999; L. M. Silva & J. G. Silva, 1992). That is, the initiator process requests any other process to take a checkpoint by broadcasting only the control message (M_c) during any invocation of the algorithm. There is no other control message used. So our algorithm ensures effective utilization of the limited wireless bandwidth. In this context, it may be noted that our algorithm needs much less number of the system messages than in (R. Koo & S. Toueg, 1987; G. Cao & M. Singhal, 2001; E. N. Elnozahy et al., 1992; R. Ahmed & A. Khaliq, 2003).
2. The second requirement about the efficient use of the mobile host's battery power is satisfied, because (1) each MH is interrupted only once by the control message M_c, as our algorithm is a single phase one. It saves time since interrupt handling time cannot be ignored. Note that in other approaches (G. Cao & M. Singhal, 2001; R. Ahmed & A. Khaliq, 2003) it is more than one; and (2) each process P_i only checks if its $c_i = 1$ in order to decide if it needs to take a checkpoint. This is the only computation that an MH is involved with while participating in the algorithm.
3. The third requirement about the efficient use of the mobile host's memory is satisfied, because the data structure used in our algorithm is very simple. Only four variables are needed by each process P_i. These are: three integer variables, viz. N_i, $counter_i$, CLK_i, and one Boolean variable c_i. The amount of data structures stated above is much less than the same in the related works (G. Cao & M. Singhal, 2001; R. Ahmed & A. Khaliq, 2003).

7. Conclusions

In this work, we have presented a non-blocking synchronous check pointing approach to determine globally consistent checkpoints. In the present work only those processes that have sent some message(s) after their last checkpoints, take checkpoints during check pointing; thereby reducing the number of checkpoints to be taken. This approach offers advantage particularly in case of mobile computing systems where both non-block check pointing and reduction in the number of checkpoints help in the efficient use of the limited resources of mobile computing environment.

Also, the presented non-blocking approach uses minimum interaction (only once) between the initiator process and the system of n processes and there is no synchronization delay. This is particularly useful for mobile computing environment because of less number of interrupts caused by the initiator process to mobile processes, which results in better utilization of the limited resources (limited battery power of mobile machines and wireless bandwidth) of mobile environment. To achieve this we have used very simple data structures, viz., three integer variables and one Boolean variable per process. Another advantage of the proposed algorithm is that each process takes its check pointing decision independently which may become helpful for mobile computing. The advantages mentioned above make the proposed algorithms simple, efficient, and suitable for mobile computing environment.

The check pointing algorithm ensures that there is no orphan message between ant two checkpoints belonging to the recovery line. However, absence of orphan messages alone cannot guarantee correctness of the underlying distributed application. To ensure correct computation all lost messages at the time of failure have to be identified and resent to the appropriate receiving processes when the system restarts. The presented recovery approach handles the lost messages using the idea of sender-based message logging to ensure correctness of computation.

8. References

Y.M. Wang, (1997). Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints, *IEEE Transactions on Computers,* vol. 46, No.4, pp. (456-468).

M. Singhal and N. G. Shivaratri. (1994). In: *Advanced Concepts in Operating Systems,* McGraw-Hill.

R. Koo and S. Toueg, (1987). Checkpointing and Rollback-Recovery for Distributed Systems, *IEEE Transactions on Software Engineering*, vol.13, No.1, pp. (23-31).

S. Venkatesan, T. T-Y. Juang, and S. Alagar, (1997). Optimistic Crash Recovery without Changing Application Messages, *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, No. 3, pp. (263-271).

G. Cao and M. Singhal, (1998). Coordinated Checkpointing in Distributed Systems, *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, No.12, pp. (1213-1225).

D. Manivannan and M. Singhal, (1999). Quasi-Synchronous Checkpointing: Models, Characterization, and Classification, *IEEE Transactions on Parallel and Distributed Systems*, vol.10, No.7, pp. (703-713).

G. Cao and M. Singhal, (2001). Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing Systems, *IEEE Transactions on Parallel and Distributed Systems*, vol.12, No. 2, pp. (157 – 172).

E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, (1992). The Performance of Consistent Checkpointing, *Proceedings. 11th Symp. of Reliable Distributed Systems.* October, 1992.

L. M. Silva and J. G. Silva, (1992). Global Checkpointing for Distributed Programs, *Proceedings, 11th Symp. of Reliable Distributed Systems*, October, 1992.

P. Jalote, (1998). Fault Tolerance in Distributed Systems. *PTR Prentice Hall*, (1994), *Addison-Wesley*, (1998).

R. Ahmed and A. Khaliq, (2003). A Low-Overhead Checkpointing Protocol for Mobile Networks, *IEEE CCECE 2003*, vol. 3, pg. (4 – 7), May 2003.

D. B. Johnson and W. Zwaenepoel, (1987). Sender-Based Message Logging, *Proceedings of 17th Intl. Symposium on Fault Tolerant Computing Systems*, Pittsburgh, 1987.

M. L. Powell and D. L. Presotto, (1983). Publishing: A Reliable Broadcast Communication Mechanism, *Proceedings of 9th ACM Symposium on Operating Systems*.

L. Alvisi and K. Marzullo, (1995). Message Logging: Pessimistic, Optimistic, and Causal, *Proceedings of 15th IEEE Intl. Conference on Distributed Computing Systems*.

B. Randell, (1975). System Structure for Software Fault Tolerance, *IEEE Transactions on Software Engineering*, vol. 1, pp. (226 - 232).

R. E. Strom, S. Yemini (1985). Optimistic Recovery in Distributed Systems, *IEEE Transactions on Software Engineering*, vol. 3, No. 3. pp. (204 – 226).

K. Venkatesh, T. Radhakrishnan, and H. F. Li, ((1987). Optimal Check Pointing and Local Recording for Domino-Free Rollback Recovery, *Information Processing Letters*, vol. 25, No. 5, pp. (295 – 304).

B. Gupta, S. K. Banerjee, and B. Liu, (2002). Design of New Roll-Forward Recovery Approach for Distributed Systems, *IEE Proceedings – Comput. Digit. Tech.*, vol. 149, No. 3, pp. (105 -112).

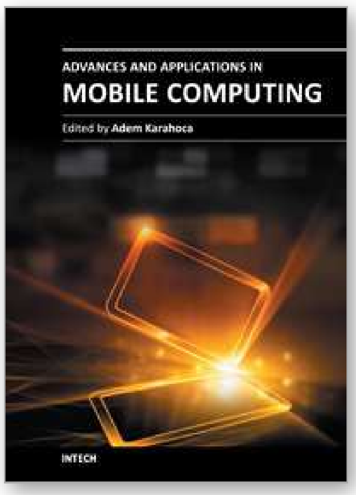
J. Tsai, S-Y Kuo., and Y-M Wang, (1998). Theoretical Analysis for Communication-Induced Checkpointing Protocols with Rollback-Dependency Trackability, *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, No.10, pp. (963 – 971).

R. Baldoni, J. M. Helway, A. Mosterfaoui, and M. Raynal, (1997). A Communication-Induced Checkpointing Protocol That Ensures Rollback Dependency Trackability, *Proc. IEEE Int'l Symp. Fault-Tolerant Computing*, pp. (68-77), 1997.

J. M. Helary, A. Mosterfaoui, R. H. B. Netzer, and M. Raynal, (2000). Communication-Based Prevention of Useless Checkpoints in Distributed Computations', *Distributed Computing*, vol. 13, No.1, pp. (29 – 43).

D. K. Pradhan and N. H. Vaidya, (1994). Roll-forward Check Pointing Scheme: A Novel Fault-Tolerant Architecture, *IEEE Transactions on Computers*, vol. 43, No. 10, pp. (1163 – 1174).

R. Baldoni, F. Quaglia, and P. Fornara, (1999). An Index-based Check Pointing Algorithm for Autonomous Distributed Systems, IEEE Transactions on Parallel and Distributed Systems, vol. 10, No. 2, pp. (181 – 192).



Advances and Applications in Mobile Computing

Edited by Associate Prof. Adem Karahoca

ISBN 978-953-51-0432-2

Hard cover, 224 pages

Publisher InTech

Published online 30, March, 2012

Published in print edition March, 2012

Advances and Applications in Mobile Computing offers guidelines on how mobile software services can be used in order to simplify the mobile users' life. The main contribution of this book is enhancing mobile software application development stages as analysis, design, development and test. Also, recent mobile network technologies such as algorithms, decreasing energy consumption in mobile network, and fault tolerance in distributed mobile computing are the main concern of the first section. In the mobile software life cycle section, the chapter on human computer interaction discusses mobile device handset design strategies, following the chapters on mobile application testing strategies. The last section, mobile applications as service, covers different mobile solutions and different application sectors.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Bidyut Gupta, Ziping Liu and Sindoora Koneru (2012). A Low-Overhead Non-Block Check Pointing and Recovery Approach for Mobile Computing Environment, Advances and Applications in Mobile Computing, Associate Prof. Adem Karahoca (Ed.), ISBN: 978-953-51-0432-2, InTech, Available from: http://www.intechopen.com/books/advances-and-applications-in-mobile-computing/fault-tolerance-in-distributed-mobile-computing-environment

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

© 2012 The Author(s). Licensee IntechOpen. This is an open access article distributed under the terms of the Creative Commons Attribution 3.0 License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

IntechOpen

IntechOpen