Journal of KIIT. Vol. 17, No. 5, pp. 21-27, May 31, 2019. pISSN 1598-8619, eISSN 2093-7571 21
http://dx.doi.org/10.14801/jkiit.2019.17.5.21

Duplicated Control Data Purging Algorithms for SBML Protocol Tolerating Temporary Communication Errors

Hee-Kwon Chin*, Jin-Ho Ahn**

This work was supported by Kyonggi University HubCon(Humanities Based Convergence) Research Grant 2014 (Project Number: 2014-001-HubCon).

Abstract

The sender-based message logging protocol tolerating temporary communication errors potentially has the complete recovery data for every message maintained in the volatile storage of its sender as well as in volatile storages of senders of its dependents. This paper presents two duplicated control data purging algorithms to expunge needless recovery data from volatile storages of senders of the dependents without requiring any extra control messages. The first algorithm satisfies this goal with a very little overhead by piggybacking an integer value on the notification and check messages about the receive sequence number of every message. The other algorithm exploits a table with piggybacking data to boost expunging needless recovery data and have enough available space of volatile storage remain long to the hilt. The experimental outcomes illustrate the two presented algorithms perform vastly better than the traditional one in terms of volatile storage availability.

요 약

일시적인 통신 오류를 포용하는 송신자 기반 메시지 로깅 프로토콜은 각 메시지를 위한 완전한 회복데이터를 메시지 송신자의 휘발성 저장소뿐만 아니라 그 메시지에 종속된 메시지들의 송신자 저장소에도 유지도록 할 수 있다. 본 논문에서 각 메시지의 종속된 메시지들의 송신자 휘발성 저장소로부터 불필요한 회복데이터를 제거하며 어떠한 추가 제어 메시지도 요구하지 않는 두 개의 중복된 제어 데이터 제거 알고리즘을 제안한다. 첫 번째 알고리즘은 초저비용으로 이러한 목적을 만족시키기 위해 각 메시지의 수신 일련번호에 대한 알림메시지와 확인 메시지에 한 정수 값만을 포함시킨다. 두 번째 알고리즘은 피기백 정보를 포함한 하나의 테이블을 이용하여 불필요한 회복데이터 제거를 가속화하고 휘발성 저장소의 충분한 가용 공간이 가능한 오랫동안 유지되도록 한다. 실험 결과는 제안한 두 알고리즘이 휘발성 저장소 가용성 측면에서 기존 알고리즘에 비해 상당히 성능이 좋다는 것을 보여준다.

Keywords
distributed system, fault-tolerance, consistent recovery, message logging, log purging

* Dept. of Law, Kyonggi University
- ORCID: https://orcid.org/0000-0002-7307-6140
** School of Computer Science & Eng., Kyonggi University
- ORCID: https://orcid.org/0000-0001-8776-5185

· Received: Apr. 04, 2019, Revised: May 27, 2019, Accepted: May 30, 2019
· Corresponding Author: Jin-Ho Ahn
School of Computer Science & Eng., Kyonggi University, 154-42 Gwanggyosan-ro, Yeongtong-gu, Suwon-si, Gyeonggi-do 16227, Korea,
Tel.: +82-31-249-9674, Email: jhahn@kgu.ac.kr

Ⅰ. Introduction

Currently, as a lot of very powerful micro-processors and high speed network hardwares are produced at low prices, large-scale distributed systems, composed of heterogeneous inter-connected nodes, may become the basis of high-performance and inexpensive parallel processing environments[1]. Therefore, long-running scientific applications on large-scale distributed systems can concurrently use all the available resources in clusters of heterogeneous computers[2][3]. However, when designing and implementing the systems for this purpose, an important problem should be considered. The problem is that as their size becomes greatly larger, their vulnerability to node failures may raise together[2][3]. Thus, they require ways to allow the applications computation to persist despite future crashes. Moreover, the techniques should not result in high overhead. With deliberately saving messages received by each process on volatile or stable storage with its checkpoints, log-based rollback recovery allows a system to be restored beyond its latest globally consistent state. This behavioral property is beneficial for the application fields with the difficulty of rolling back to the earlier state. Among this kind of techniques, sender-based message logging(SBML)[4]-[7] allows each message to be saved on its sender's volatile storage with no synchronous logging on stable storage. Therefore, it's normal operation overhead is much smaller than that of the receiver-based message logging(RBML)[8].

Although it has several advantages like being completely implemented as a software solution and supporting lower cost fault-tolerance compared to RBML, it holds two weaknesses in case there occur some temporary communication errors [4]. First, as the errors potentially have some messages received be partially logged, but their subsequently received messages, fully logged, the recovery process of the original SBML[5]-[7] may continue to perform no longer in case of failures of their receivers. Next, if several application messages may not be totally logged due to the transitory communication errors, this situation may make every send function call invoked after the last received message postponed till the termination of their fully logging procedures is notified by their receivers. To overcome this kind of shortcoming, our previous work presented a consistent SBML protocol enabling every process to piggyback a little control data for partially logged messages on each control message. In this case, the transmitter of every previously sent message can get the receive sequence number(rsn) granted for the message. But, it potentially have the complete recovery data for every message maintained both on the volatile storage of its sender and on volatile storages of senders of its dependents. This paper proposes selective stable message log purging algorithms to expunge needless recovery data from volatile storages of their dependents' senders without resulting in any extra control messages. This beneficial feature can be attained by piggybacking different size of data about receive sequence numbers with no extra messages for logging. The first algorithm only piggybacks an integer value with no extra messages for logging to impose minimum overhead on network link. The second exploits a table with piggybacking data to boost expunging needless recovery data and have enough available space of volatile storage remain long to the hilt.

Ⅱ. The Proposed Algorithms

The original SBML[5]-[7] has the following two problems.

- Potential Inconsistency problem and high recovery cost: its recovery process may not continue to go ahead when temporary transmission errors between failed and other processes occur and so should void rsns of completely logged messages and re-execute a large number of completely logging operations.

- Failure-free performance degradation: every send function call invoked after the last received message has to be postponed till the termination of their fully logging procedures is notified by their receivers.

To address these issues, our previous SBML protocol in [4] was proposed that piggybacks rsns of all the unstable messages on each notification message for a message received right before to its sender. So, this beneficial feature can ensure not only consistent recovery, but also processing postponed messages scheduled to be transmitted much earlier with a little extra cost despite temporary communication faults occurrence. The protocol has every process hold the variables as follows.

- Ssn_i: the send sequence number(SSN) of the last message P_i has sent before.
- Rsn_i: the receive sequence number(RSN) of the last message P_i has deliver before.
- $SsnVector_i$: a table where $SSNVt_i[j]$ is the ssn of the latest message from process P_j that P_i delivered to the application.
- $Sendlg_i$: a set keeping e(rcvr, ssn, rsn, data) for every message P_i has sent. Here, element e is the recovery data of a message consisting of the four fields, the identifier of receiver(RID), SSN, RSN and data of the message.
- $UMLg_i$: a set keeping e(sndr, ssn, rcvr, rsn) from recovery data for all unstable messages included in the notification message to P_i. Here, e is the recovery data of each unstable message, where the four fields are composed of its sender's identifier(SID), SSN, RID and RSN.
- $stableRSN_i$: the RSN of the last message delivered to P_i or saved when checkpointing. It is exploited for indicating which messages are currently stable.

In the protocol, on delivering the rsn value of message m to q from p, it piggybacks recovery data about all unstable received ones before m and after its last checkpoint on the notification message. A message is called unstable if its dependents don't currently have complete information about its fully logging. In contrast, a message is named, stable, if its property is opposite to that of unstable message. Moreover, the recovery data of each piggybacked unstable message is composed of three fields, its RSN, SSN, and SID. As q obtains the notification message, it should hold the recovery data for the unstable messages attached to the notification message in its memory buffer while m's rsn is reflected into its own log element. The check message for the RSN receipt of m triggers invoking every send function call requested after the unstable messages, but postponed till the termination of their fully logging procedures is notified by their receivers. For instance, when p2 recognizes fully logging of m3 on p4's volatile buffer in Fig. 1, it makes every postponed send function call invoked for actually transmitting out it because the rsn values of all the three messages obtained from p4 can be delivered to p2 during recovery unlike the original SBML. If p tries to save a local checkpointed state, it also enables every postponed send function calls before this state to start executing.

However, it may save not only the complete recovery data for every message on its sender's volatile storage, $Sendlg_{sndr}$, but also on the volatile storages of senders of its dependents, $UMLg_P$. For instance, after the check messages about the receipt of m1's and m2's RSNs can be obtained by p2 from p1 and p3 in Fig. 1, the recovery data for every message exists on both volatile storages of its sender and its successor's sender. For this purpose, it needs cost-effective log purging algorithms to expunge needless recovery data from volatile storages of senders of message dependents.

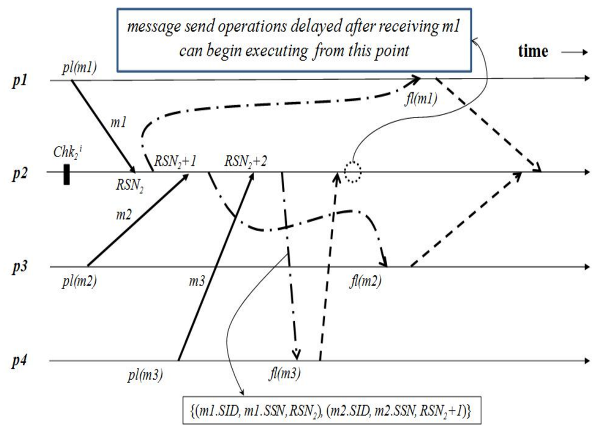
message send operations delayed after receiving m1 can begin executing from this point
time
p1
pl(m1)
fl(m1)
m1
Chk2^1
p2
RSN2+1
RSN2+2
RSN2
m2
p3
pl(m2)
m3
fl(m2)
p4
pl(m3)
fl(m3)
{(m1.SID, m1.SSN, RSN2), (m2.SID, m2.SSN, RSN2+1)}

Fig. 1. A case depicting shortcomings of [4]

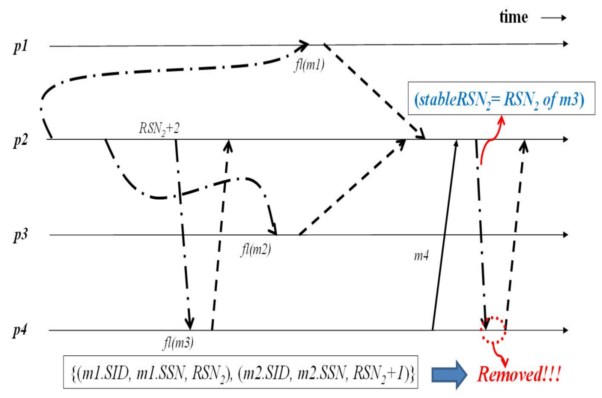

Fig. 2. A case depicting effectiveness of L-PURGER

For this purpose, one of two stable message log purging algorithms requiring no extra control message is first introduced, called L-PURGER, enabling a process to piggyback an integer value stableRSN$_p$ on each notification or acknowledgment control message. Here, stableRSN$_p$ is the RSN of the last message which was kept in its sender's volatile memory and which has been received and seen by p. For instance, if a new message m4 is delivered to p2 from p4, p2 increases its RSN and reply with it with stableRSN$_2$ to p4, which can remove the recovery data of m1 and m2 from its volatile storage in Fig. 2.

Also, if p4 has previously received several messages and p2 is the transmitter of their dependents, p2 can expunge the duplicated recovery data about the messages from the volatile storage of p2, UMLg$_2$, after m4's RSN acknowledgement with stableRSN$_4$ is delivered to p2 by p4. This case shows our proposed algorithm requires no additional interactions among processes. The first is designed for imposing minimum piggybacking overhead on network links.

The second stable message log purging algorithm, named V-PURGER, exploits a table with piggybacking data, lRSNs$_P$, for incurring no extra message and forced checkpoint. With this table, this algorithm may significantly boost expunging needless recovery data and have enough available space of volatile storage remain long to the hilt compared with the first one L-PURGER.

V-PURGER can be combined with the protocol in [4] like in Fig. 3. The two algorithms enable their users to control piggybacking overhead depending on their preference and network condition.

III. Performance Evaluation

Brief experiments are performed in this paper to compare performance of our two algorithms, L-PURGER and V-PURGER, and that of the traditional one, T-PURGER[6], using a simulation tool[9]. For this purpose, the first performance index is exploited to measure the effectiveness of the presented algorithms; the mean time required till having the volatile memory used on logging for every process full-packed(T_{bfull}). This factor is the most important metric for this comparison because this kind of inter-message dependency data piggybacking algorithm is designed to potentially have the storage for logging messages available till as late as possible. The evaluation factor T_{bfull} is examined under the condition that the three algorithms execute with no duplicated control messages and checkpoints. The other factor is exploited for measuring extra checkpointing overheads unavoidably incurred despite each algorithm's effort; the average number of additional checkpoints that should be forced to take due to the unavailability of the volatile storage(T_{addckt}).

```
Module Msg-Send(data, rcvr) at $P_{sndr}$
  increment $Ssn_{sndr}$ by one ;   assign $Ssn_{sndr}$ to data ;   send m(data, $Ssn_{sndr}$) to $P_{rcvr}$ ;
  $Sendlg_{sndr} \leftarrow Sendlg_{sndr} \cup \{(rcvr, Ssn_{sndr}, -1, data)\}$ ;

Module Msg-Recv(m(ssn, data, sndr)) at $P_{rcvr}$ (including Part 1)
  if($SsnVector_{rcvr}$[m.sndr] < m.ssn) then
    $Rsn_{rcvr} \leftarrow Rsn_{rcvr} + 1$ ;     $SsnVector_{rcvr}$[m.sndr] $\leftarrow$ m.ssn ;
    for all $e \in RSNVector_{rcvr}$ st (e.rsn > $stableRSN_{rcvr}$) do
      UnstableMsgs $\leftarrow$ UnstableMsgs $\cup$ {(e.sid, e.ssn, rcvr, e.rsn)} ;
    send return(m.ssn, $Rsn_{rcvr}$, $lRSNs_{rcvr}$) with UnstableMsgs to $P_{m.sndr}$ ;
    $RSNVector_{rcvr} \leftarrow RSNVector_{rcvr} \cup \{(m.sndr, m.ssn, Rsn_{rcvr})\}$ ;
    delay all the send message operations generated after having received m ;
    deliver m.data to its corresponding application ;
  else
    find $\exists e \in RSNVector_{rcvr}$ st ((i.SID = m.sndr) ^ (i.SSN = m.ssn)) ;
    for all $e \in RSNVector_{rcvr}$ st ((e.rsn < i.RSN) ^ (e.rsn > $stableRSN_{rcvr}$)) do
      UnstableMsgs $\leftarrow$ UnstableMsgs $\cup$ {(e.sid, e.ssn, rcvr, e.rsn)} ;
    send return(m.ssn, i.RSN, $lRSNs_{rcvr}$) with UnstableMsgs to $P_{m.sndr}$ ;

Module RSN-Rcvr(return(ssn, rsn, $lRSNs_{rcvr}$, rcvr, UnstableMsgs)) at $P_{sndr}$ (including Parts 2
and 3)
  find $\exists e \in Sendlg_{sndr}$ st ((e.rid = return.rcvr) ^ (e.ssn = return.ssn)) ;
  e.rsn $\leftarrow$ return.rsn ; $\forall$i: $lRSNs_{sndr}[i] \leftarrow$ max($lRSNs_{sndr}[i]$, $lRSNs_{rcvr}[i]$) ;
  $UMLg_{sndr} \leftarrow UMLg_{sndr} \cup$ return.UnstableMsgs ;
  send ack(return.rsn)   with $lRSNs_{sndr}$ to $P_{return.rcvr}$ ;
  call Module Remove-LogForstableMsgs( ) at itself ;

Module RSN-Ack(ack(rsn, sndr, $lRSNs_{sndr}$)) at $P_{rcvr}$ (including Part 4)
  if($stableRSN_{rcvr}$ < ack.rsn) then
    allow all the send message operations delayed before receiving the message
    whose rsn value is (ack.rsn+1) to begin executing ;
    $lRSNs_{rcvr}[rcvr] \leftarrow stableRSN_{rcvr} \leftarrow$ ack.rsn ;
    $\forall$i: $lRSNs_{rcvr}[i] \leftarrow$ max($lRSNs_{sndr}[i]$, $lRSNs_{rcvr}[i]$) ;
    call Module Remove-LogForstableMsgs( ) at itself ;

Module Checkpointing() at $P_i$
  take its local checkpoint with ($Rsn_i$, $Ssn_i$, $SsnVector_i$, $Sendlg_i$, $UMLg_i$)
  on the stable storage ;
  allow all the send message operations delayed before this checkpoint to begin executing ;
  $lRSNs_i[i] \leftarrow stableRSN_i \leftarrow Rsn_i$ ;     make $RSNVector_i$ an empty set ;

Module Remove-LogForstableMsgs( ) at $P_i$
  for all pid $\in$ P st (pid $\neq$ i) do
    for all $e \in UMLg_i$ st ((e.rid = pid) ^ (e.rsn $\leq$ $lRSNs_i$[pid])) do
      $UMLg_i \leftarrow UMLg_i - \{e\}$ ;
```

Fig. 3. Our SBML protocol including V-PURGER

A general network based system composed of 10 nodes inter-connected is simulated. Each process performs its own task on one node and it is supposed to start and terminate its execution at the same time for simplicity of experiment. The message transmission capacity of a link in the network is 100Mbps and its propagation delay is 1ms. For the simulation, suppose that the size for every message ranges from 1KB to 1MB and the volatile storage size for saving messages is 128MB per process. Every process takes normal checkpointing with a checkpoint interval following an exponential distribution with a mean T_{nc}=300 seconds. In addition, the mean message sending rate, $T_{interval}$, follows an exponential distribution.

Fig. 4 illustrates the mean time of the three algorithms spent till having the volatile memory used on logging for every process full-packed for the specified range of the $T_{interval}$ values. As $T_{interval}$s of all the algorithms arise in Fig. 4, their corresponding T_{bfull}s also become higher. The outcome comes from the behavioral feature that as application messages are gradually transmitted slower, their message log size also increases at a lower rate. However, as we can expect, T_{bfull} of algorithm T-PURGER is significantly faster than the two algorithms L-PURGER and V-PURGER. Similarly, V-PURGER vastly surpass L-PURGER in T_{bfull}. Specifically, as $T_{interval}$ increments accordingly, the increasing rate of V-PURGER becomes much higher than that of L-PURGER. The advantage of our algorithms comes from their following desirable feature: with these algorithms, each process p can willingly and locally expunge needless recovery data from the volatile storage by only carrying an integer value or a table with piggybacking data on each control message whereas the traditional algorithm does not so.

Fig. 5 illustrates the average number of additional checkpoints to be forced to take triggered by the unavailability of the volatile storage(T_{addckt}) for the various $T_{interval}$ values. With this result, it may be recognized that T_{addckt}s of the three algorithms arise when $T_{interval}$ becomes gradually lower.

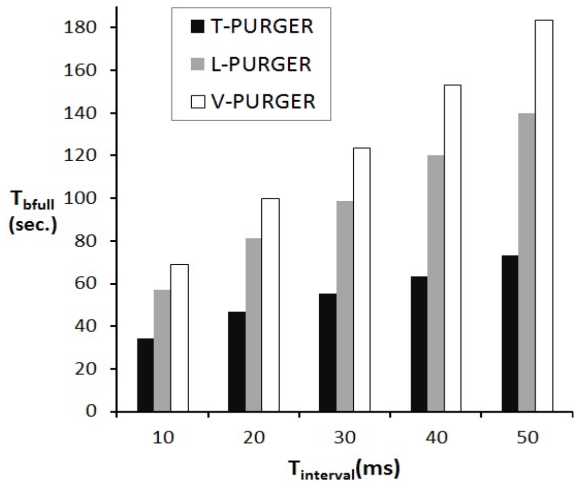

Fig. 4. T_{bfull} according to $T_{interval}$

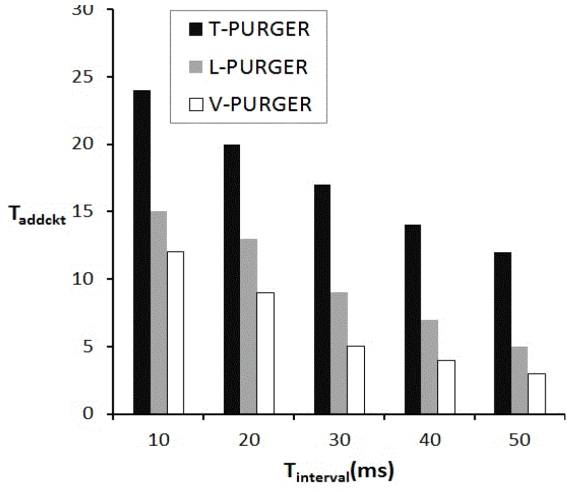

Fig. 5. T_{addckt} according to $T_{interval}$

This outcome results from the following reason; as processes often disseminate messages, the speed of message log stacking up on the volatile storage may be much faster despite effectiveness of local message purging. In the figure, L-PURGER's T_{addckt} is much lower than that of T-PURGER, reducing up to 58%, which illustrates effect of local garbage collection gained by inter-message dependency data piggybacking. Also, V-PURGER performs better than L-PURGER in terms of T_{addckt} because V-PURGER utilizes table data structure for exchanging dependency data about message stability. However, as the number of processes becomes larger, the piggybacking overhead of V-PURGER may be a performance bottleneck compared with L-PURGER.

Therefore, our proposed algorithms L-PURGER and V-PURGER are fairly effective for making enough available space of volatile storage remain long to the hilt while minimizing network overhead compared with the traditional one. In addition, depending on the scale of the system and its network characteristics, one of our proposed algorithms performs better than the other.

Ⅳ. Conclusion

In this paper, two redundant stable message log purging algorithms L-PURGER and V-PURGER were proposed requiring no extra control message interaction. L-PURGER satisfies this goal with a very little overhead by piggybacking an integer value in the notification and check messages about the RSN for every message whereas V-PURGER exploits a table with piggybacking data to boost expunging needless recovery data and potentially have enough available space of volatile storage remain long to the hilt. The experimental outcomes illustrate the two algorithms perform vastly better than the traditional one in terms of volatile storage availability while they have their respective strengths and weaknesses against each other. Therefore, it is believed that the two algorithms can greatly enhance availability of the volatile memory each process holds with low cost if they are applied into the protocol in[4].

For future research, our SBML protocol with the proposed algorithms will be refined to enable their users to control piggybacking overhead depending on their preference and network condition.

References

[1] S. Di, L. Bautista-Gomez, and F. Cappello, "Optimization of multi-level checkpoint model with uncertain execution scales", In Proc. of the International Conference for High Performance Computing, Networking, Storage, and Analysis, New Orleans, LA, USA, pp. 907-918, Nov. 2014.

[2] L. Bautista-Gomez, T. Ropars, N. Maruyama, and S. Matsuoka, "Hierarchical clustering strategies for fault tolerance in large scale HPC systems", In Proc. of the IEEE International Conference on Cluster Computing, Beijing, China, pp. 355-363, Sep. 2012.

[3] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. J. Dongarra, "Post-failure recovery of MPI communication capability: design and rationale", The International Journal of High Performance Computing Applications, Vol. 27, No. 3, pp. 244-254, Jun. 2013.

[4] J. Ahn, "Lightweight Consistent Recovery Algorithm for Sender-based Message Logging in Distributed Systems", IEICE Transactions on Information and Systems, Vol. E94-D, No. 8, pp. 1712-1715, Aug. 2011.

[5] P. Jaggi and A. Singh, "Log based recovery with low overhead for large mobile computing systems", Journal of Information Science and Engineering, Vol. 29, No. 5, pp. 969-984, Sep. 2013.

[6] D. Johnson and W. Zwaenpoel, "Sender-based Message Logging", In Proc. of the 7th International Symposium on Fault-Tolerant Computing, pp. 14-19, Jul. 1987.

[7] H. Meyer, D. Rexachs, and E. Luque, "Hybrid Message Pessimistic Logging. Improving Current Pessimistic Message Logging Protocols", Journal of Parallel and Distributed Computing, Vol. 104, No. C, pp. 206-222, Jun. 2017.

[8] B. Yao, K. Ssu, and W. Fuchs, "Message logging in mobile computing", the 29th International Symposium on Fault-Tolerant Computing, pp. 14-19, Nov. 1999.

[9] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, and H. Y. Song, "Parsec: A Parallel Simulation Environments for Complex Systems", IEEE Computer, Vol. 31, No. 10, pp. 77-85, Oct. 1998.

Authors

Hee-Kwon Chin

1998 : Ph.D. in Dept. of Law, Korea University
2003 ~ Present : Professor of Dept. of Law, Kyonggi University
Research Interests : Philosophy of Law, Data Privacy Law

Jin-Ho Ahn

2003 : Ph.D. in Dept. of Computer Science & Eng., Korea University
2003 ~ Present : Professor of School of Computer Science & Eng., Kyonggi University
Research Interests : Distributed & Parallel Computing, Cloud Computing, CPS