Real-Time Cheat-Free Gaming on the Basis of Time-Stamp Service

Shunsuke Mogaki[†], Masaru Kamada[†], Tatsuhiro Yonekura[†], Shusuke Okamoto[††], Yasuhiro Otaki[†] and Mamun Bin Ibne Reaz[†††],

[†]Graduate School of Computer and Information Sciences, Ibaraki University, 4-12-1, Nakanarusawa, Hitachi, Ibaraki 316-8511, Japan
[††]Faculty of Science and Technology, Seikei University, Musashino, Tokyo 180-8633, Japan
[†††]Department of Electrical and Computer Engineering, International Islamic University Malaysia, Jalan Gombak, 53100 Kuala Lumpur, Malaysia

Summary
A cheat-proof protocol for real-time gaming is proposed under the assumption that time-stamp servers issue serially numbered time stamps honestly and are available near every player, i.e., they exist everywhere in the Internet. With this protocol, each player sends its action to the other player and also sends its hash to the nearest time-stamp server. The time-stamp server sends the signed hash with the time and a serial number back to the player. The actions are checked to verify that they are compatible with the hashes, and the signed hashes are checked to verify that they have the correct time and the serial numbers are contiguous. The only latency in this protocol is the travel time of the packet from one player to another. In comparison with other existing protocols, we confirm that the proposed protocol is as fast as and more secure than the fair synchronization protocol, the fastest existing protocol.

Key words:
real-time network gaming, cheat-proofing, time-stamp service

1. Introduction

Network games have been suffering from two major kinds of cheating. One targets the games where the client software controls disclosure of secrets to the player. Cheaters tamper the client software in there favor, for example, to make walls transparent so that they can see enemies behind the walls. Laurens et al. [1] extensively studied how to detect cheats of this kind by analyzing player behaviors. The other major cheat is late commitment which targets all the real-time games. Cheaters send their action after receiving other players' action and blame the network latency of the delayed arrival of their actions.

The existing protocols against late commitment in network gaming have been created for a peer-to-peer setting and do not assume any trusted third parties. In fact, for those working on cryptographic protocols, it is even considered to be a scientific ``cheat" to assume a trusted third party.

The lockstep protocol [2] is the most successful cryptographic protocol to realize a fair environment for gaming without trusting any third parties. Unfortunately, the lockstep is tied to a game clock and not the real-world clock. The game clock can be stopped by any player, and so gaming by the lockstep protocol is never actually in real time. The asynchronous synchronization [3] allows for omitting the lock-step sessions between players who are remote from each other in the game field since they would not have any influence on each other. But neighbors in the virtual space are not always neighbors in the network.

In an attempt to make a real-time version of the lockstep protocol, several researchers have proposed modifications under the assumptions that the players decide their actions in synchronization to the real-world clock and that the network latency is known [4][5]. But players can easily cheat each other by reporting a longer latency than the real one. We may have to conclude that a cheat-proof protocol for real-time gaming must assume some trusted timing supervisors.

The fair synchronization protocol [6] is the first protocol where distributed timing supervisors, named pulsers, are employed to synchronize the game clock to the real-world clock. The pulsers share a series of common keys, each of which is assigned to game-clock timing. Each player sends its action to the nearest pulser by a deadline. The pulser sends the action to the other players in encrypted form. Because the encrypted actions may reach the players at different times, each player receives a common key to decrypt the actions at the same time from the nearest pulser. It is beneficial that players are forced to decide their actions by a common time and are allowed to know the other players' actions at a common time. But, the players have to disclose private information such as their actions to the pulsers. In that sense, the pulsers constitute a

Manuscript received January 5, 2009
Manuscript revised January 20, 2009

distributed trusted third party. However, the pulsers are capable of tampering with the actions.

Distributed timing supervisors have only to force players to fix actions by a common deadline. Players do not have to disclose their actions to the supervisors. It suffices for the proof of a player's honesty that he keeps proper time stamps on his actions.

In this paper, a cheat-proof protocol for real-time gaming is proposed under the assumption that we trust time-stamp servers only with respect to their job in time-stamping and that they are available everywhere in the Internet.

The rest of this paper is organized as follows: Section 2 gives assumptions, procedure and analyses of the proposed protocol. In Section 3, we compare it with four existing protocols. Section 4 will give a summary and a future work.

2. Proposed Protocol

2.1 Assumptions

The time-stamp service is becoming a common infrastructure of the Internet. It guarantees that the data exist at the time of stamping and that the data have not been tampered since the stamped time. Without trusting the time-stamp servers, it would not be possible to have such a fundamental service as an electronic notary over the Internet [7]. In Japan, for example, the time business accreditation center [8] authorizes time-stamping companies. Several companies are already in business.

The time-stamp servers must be trusted in order to force the players to take actions synchronized to the real-world clock, $t_i = i \tau t$, (i = 1,2,3…), where $\tau t > 0$ is the frame interval. More precisely, we make the following assumptions:

- Time-stamp servers honestly issue time stamps comprising signed hashes of data with the time and serial numbers.
- The time-stamp servers are available near each player. The packet traveling time from a player to the nearest time-stamp server, ３, will be small.

2.2 Procedure

For simplicity in explanation, we assume there are only two players without loss of generality. Multiplayer cases are superpositions of cambinatorid two-player relationships.

In the proposed protocol, each player sends its action to the other player and simultaneously sends its hash to the nearest time-stamp server. The time-stamp server sends back to the player the signed hash, which contains the data, signing time and serial number *i*. The signature is undeniable evidence of the action. Nobody can delete or insert any actions in the sequence of evidences since the serial numbers should be contiguous. The players exchange all the signed hashes to verify the actions made in the game.

The detailed procedure is illustrated in Fig.1 and performed as follows:

1. Both players agree on and share a common hash function *H*.
2. Players A and B decide their actions A_i and B_i, respectively, at time t_i. At the same time, they send their hashes, $H(A_i)$ and $H(B_i)$, along with *i*, to the nearest time-stamp servers.
3. The time-stamp servers send back the signed hash $T(H(A_i), t_i^A, i)$ and $T(H(B_i), t_i^B, i)$ to players A and B, respectively; the signed hash contains data, the signing time t_i^A and the serial number *i*. Here, t_i^A and t_i^B represent the time when the time-stamp servers receive $H(A_i)$ and $H(B_i)$, respectively.

Steps 2 and 3 are iterated concurrently for each player during the game.

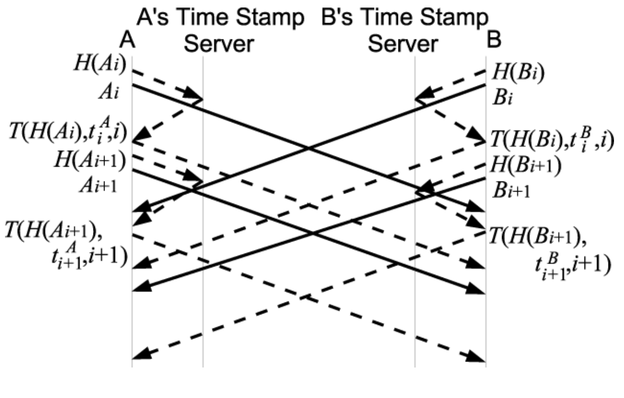

Fig. 1 Proposed Protocol.

The players exchange their signed hashes for verification of the actions made during the game to identify possible cheats. As soon as player A receives the hash from player B, A can start its verification process in a parallel thread that runs slightly behind the real time. Of course, this verification can also be done as a batch job after the end of gaming. Player A verifies B_i against $H(B_i)$ and also verifies the signature $T(H(B_i), t_i^B, i)$ against $H(B_i)$, t_i^B and i. If $t_i^B \mathrel{3} ! t_i + 3$, A can accuse B of cheating by delayed action.

2.3 Latency and Frame Interval

In the proposed protocol, each player sends his own action directly to the other player. Therefore, the latency is only the packet travel time l from one player to another. No one can make this time shorter, so the proposed protocol has minimum latency. Player can always verify if their hashes are correctly stamped.

The frame interval can be made as short as required in principle. In practice, the frame rate is limited by the throughput of the time stamp servers.

The hash can be verified by the delay 2 3 relative to the receipt of action.

2.4 Security

The actions are hashed so that the time-stamp servers obtain no information about the actions. This helps protect the players' privacy.

A corrupted time-stamp server may give retroactive stamps or sign several possible actions with the same serial number in favor of a specific player. In either case, the proposed protocol collapses.

There has been a countermeasure against the retroactive time stamping. That is called the Time-Stamp Authority (TSA) grid [9], which is a union of time-stamp servers that audit one another. In the TSA grid, a time-stamp server sends its time stamp to another server to get another time stamp. The first time-stamp server proves its honesty by the two-fold time stamp having a sufficiently small time difference. In this case, the malicious player has to bribe the two time-stamp servers at the same time. By increasing the number of time-stamp servers that make multifold time stamps, we can make bribery as difficult as required.

In addition to the above auditing capability of the TSA grid, we shall also include the capability of detecting different actions with the same serial number. Wrongful time stamps may not be detected unless some of them happened to be passed to the same honest time stamp server. But we can reward corrupted time stamp servers with a severe penalty so that they would not dare to commit crimes.

It is possible that player A commits an action correctly but intentionally delays its transmission. In this case, player B has to decide B_{i+n} (for large n) without knowing A_i. If player B receives A_i later than expected, Player B is encouraged to take a countermeasure by delaying the disclosure of its action, too. Then the game becomes "blind shooting." But the players stay in an equal state. Besides, by including this countermeasure in the protocol, we can deter players from attempting an intentional delay because the cheater does not gain any advantages while the gaming becomes tedious.

Another possible countermeasure is to let each player verify a hash before sending the next action. Then any cheat can be detected immediately in the sense of game clock. But the frame interval must be as long as $l + 2\,3$ even in the best case. A next frame might never come in case a malicious player stops sending the hash. So this modification makes the protocol fall back to a non real-time one.

As long as we keep the assumptions regarding the time-stamp servers, we have the same security, latency and frame interval for the gaming among three players or more except that we have more possibility of having a malicious player within the increased number of players. It is not a technical but social issue that several players may join hands for entrapping a player.

If a player has only a remote time-stamp server, we have a long 3 . The time lag $l + 2\,3$ for verifying actions will be long accordingly. Besides, we have to allow for a longer time deviation of time stamps. So we have to limit the players within those having a time-stamp server near by.

It would improve the security for the players to have their actions time-stamped by several common time-stamp servers that they trust. In this case, however, we will have a long 3 comparable with l. For the real-time gaming, players must trust near-by time-stamp servers.

3. Comparison

The following is a comparison of the latency, frame interval and security of the proposed protocol with typical cheat-proof protocols, such as the lockstep protocol [2], pipelined lockstep protocol [4], sliding pipeline protocol [5] and fair synchronization protocol [6].

Table 1 shows latency and frame interval and table 2 shows advantage and draw back.

Lockstep Protocol

In synchronization to the game clock i, (i = 1,2,3…), each player sends a commitment of his action and waits for the commitment from the other player before unveiling his action. It should be noted again that the game clock is not tied to the real-world clock.

The detailed procedure is illustrated in Fig.2 and performed as follows:

1. The players agree on and share a common hash function H.
2. Players A and B decide their respective actions A_i and B_i at time i of the game clock and exchange hashes $H(A_i)$ and $H(B_i)$ as the commitment of their actions.
3. Upon receiving the hash from the counterpart, the players unveil their actions A_i and B_i to each other.

Steps 2 and 3 are iterated during the game session.

Table 1: latency and frame interval

protocol	latency	frame interval
lockstep protocol	$2l$	$2l$
pipelined lockstep protocol	$2l$	$\frac{2l}{p}$
sliding pipelined protocol	$2l$	$\frac{2l_i}{p_i}$
fair synchronization protocol	$l + 3$	$\frac{l}{p_i} + 2\ 3$
proposed protocol	l	limited by the throughput of the time stamp servers

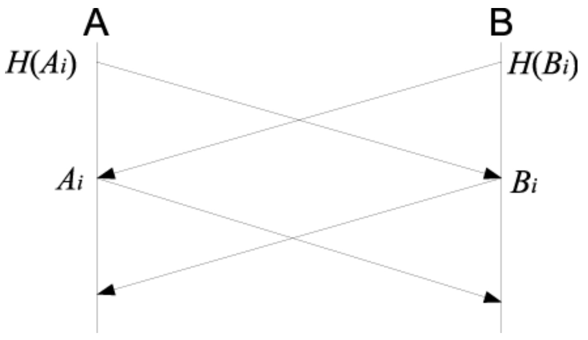

Fig. 2 Lockstep Protocol

In the lockstep protocol, the latency is the packet round-trip time $2l$, which is the time for the action of one player to reach the other. The frame interval cannot be shorter than the packet round-trip time $2l$. In contrast, with the proposed protocol, the latency is only l and the frame interval can be made as short as required.

The actions taken by the players are unknown to each other until the actions are committed. Therefore, this protocol is completely cheat-free, as is the proposed protocol. But, in this protocol the game clock remains stopped until players complete exchanging their commitments or actions. A malicious player can gain time to think about the next action as long as he desires simply by withholding his decision.

Pipelined Lockstep Protocol

In the pipelined lockstep protocol, players are assumed to take an action at time $t_i = i\Delta t$ of the real-world clock. The locksteps are pipelined in order to make the frame interval $\Delta t = t_{i+1} - t_i$ shorter than the network latency l, which is supposed to be known to the players.

The detailed procedure is illustrated in Fig.3 and performed as follows:

Table 2: advantage and drawback

protocol	advantage	drawback
lockstep protocol	completely cheat-free	not real time
pipelined lockstep protocol	high speed	allow late commitment
sliding pipelined protocol	pipeline size adapted to network latency	allow late commitment
fair synchronization protocol	Pulsers control the disclosure timing	Pulsers are completely trusted. They know actions in the plain text
proposed protocol	Time-stamp servers supervise only the time of action	Time-Stamp Servers are trusted with respect to stamp time. They know only the hash of actions

1. Given the pipeline size p, the players agree on the frame interval Δt such that $\Delta t ⊪ \frac{m}{q}!$and a common hash function H.
2. For $i = 1,2,\ldots,p$, players A and B decide their respective actions A_i and B_i at time t_i and send their respective hashes $H(A_i)$ and $H(B_i)$ to each other.
3. For $i > p$, player A decides the action A_i and sends its hash $H(A_i)$ along with the unveiled action A_{i-p} as soon as A receives the hash $H(B_{i-p})$ from player B. Player B does the same.

In the pipelined lockstep protocol, the latency remains the same as that in the lockstep protocol. But, unlike the lockstep protocol, players do not wait for commitments from the other player. The frame interval is $\frac{2l}{p}$ and can be made as short as required by increasing the pipeline size p.

The large p, however, degrades the security. If player A is malicious and B is honest, A can pretend to suffer from a large latency l despite the fact that A receives packets from B almost immediately.

Then A will receive the unveiled action B_{i-p} and new hash $H(B_i)$ at t_i, but B will not receive A_{i-p} and $H(A_i)$ until t_{i+p}. Therefore, A can take advantage of the information B_{i-2p+1}, $B_{i-2p+2},\ldots,B_{i-p}$ to decide his next action A_i, while B is unaware of the information $A_{i-2p+1},A_{i-2p+2},\ldots,A_{i-p}$. In this situation, player B has the right to appeal but is unable to show any evidence for the late arrival of B_{i-2p+1},$B_{i-2p+2},\ldots,B_{i-p}$. Thus, player A is able to insist his innocence by blaming the varying network condition.

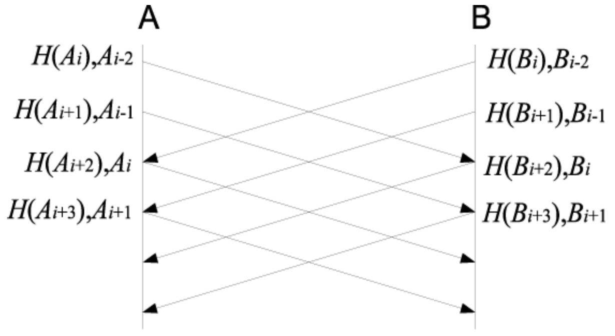

Fig. 3 Pipelined Lockstep Protocol (for the case $p = 2$).

Sliding Pipeline Protocol

The sliding pipeline protocol incorporates a means to estimate the variable network latency. The players are assumed to take an action at nonuniform instances t_i, where the frame interval $\Delta t_i = t_{i+1} ≗ t_i$ is variable.

The detailed procedure is illustrated in Fig.4 and performed as follows:

1. By exchanging several packets, players A and B estimate the network latency as l_1^A and l_1^B, respectively. Then, they exchange the estimated network latency to decide the initial latency $l_1 = \max(l_1^A, l_1^B)$.
2. Given the initial pipeline size p_1, the players agree on the frame interval Δt_1 such that $\Delta t_1 ⊪ \frac{m_2}{q_2}!$ and a common hash function H.
3. For $i = 1,2,\ldots,p_1$, the pipeline size p_i, the latency l_i and the frame interval Δt_i are set to be p_1, l_1 and Δt_1, respectively. Players A and B decide their respective actions A_i and B_i at time t_i and send their hashes $H(A_i)$ and $H(B_i)$ to each other.
4. For $i > p_1$, player A decides the action A_i and sends its hash $H(A_i)$ along with the unveiled action $A_{i-p_{i-1}}$ and new estimated network latency l_i^A as soon as A receives the hash $H(B_{i-p_{i-1}})$ from player B. Player B does the same. The updated latency is $l_i = \max(l_i^A, l_i^B)$, which is given to players A and B. According to l_i, the pipeline size p_i, and the frame interval Δt_i, such that $\Delta t_i ⊪ \frac{l_i}{p_i}$, are updated.

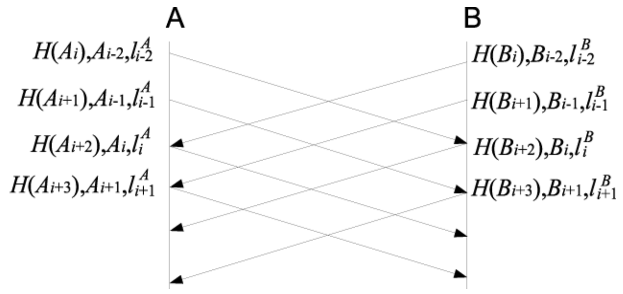

Fig. 4 Sliding Pipeline Protocol (for the case $p = 2$).

In the sliding pipeline protocol, the latency is supposed to be variable but remains almost the same as that for the lockstep protocol. The frame interval is $\frac{2l_i}{p_i}$ and can be made as short as required by increasing the pipeline size p_i.

This protocol may cope with variable network congestion. But the network latency is estimated by trusting the departure time of the packets, which can be tampered with by the sender. Therefore, the sliding technique does not provide a remedy for cheating with respect to time.

Fair Synchronization Protocol

Unlike the above three protocols, the fair synchronization protocol employs distributed supervisors. This protocol is similar to the proposed protocol in that the timing supervisors are assumed.

These supervisors, named pulsers, control the times when the actions are decided and the times when they are disclosed to the other players. In order to control the disclosure timing, a series of cryptographic keys are employed each of which is unique to the time *i*. The keys are distributed to all the pulsers prior to game session. Each player is supervised and supported by a pulser in its neighborhood reachable by the latency 3.

The communications are made in a pipeline to shorten the frame interval. The pulsers decide a variable pipeline size p_i according to the maximum latency l from pulsers to the farthest player so that all the players can receive actions made at t_i by the time t_{i+p_i}.

The detailed procedure is illustrated in Fig.5 and performed as follows:

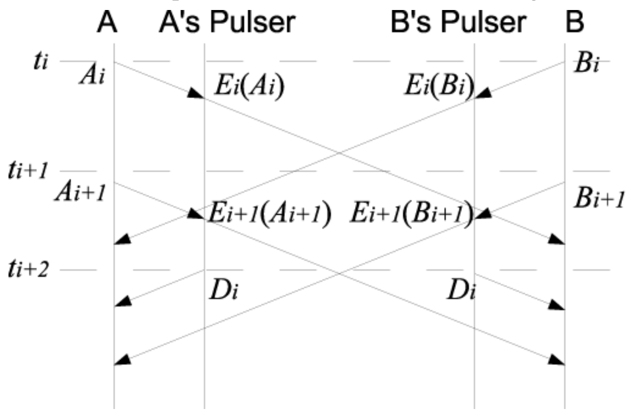

Fig. 5 Fair Synchronization Protocol (for the case $p_i = 2$).

1. Players A and B decide their respective actions A_i and B_i at t_i. Player A sends A_i to the nearest pulser. Player B does the same.

2. A's pulser encrypts A_i into $E_i(A_i)$ and sends this $E_i(A_i)$ to player B. Likewise, B's pulser sends $E_i(B_i)$ to A.

3. If A's pulser has received A_i by the time $t_i + 3$, the pulser sends the decryption key D_i so that it reaches A just before t_{i+p_i}. Likewise, B's pulser serves player B.

4. Player A decrypts $E_i(B_i)$ to know B_i after receiving the decryption key D_i from A's pulser at t_{i+p_i}. Player B does the same.

The latency is $l + 3$. The frame interval Δt_i can be as short as $\frac{l}{p_i} + 2\,3$. By increasing the pipeline size p_i, the frame interval can be shorten to be $2\,3$.

In a sense, the fair synchronization protocol has been constructed by making many copies of a trusted server and deploying them near the players. Players send their actions to the pulser in plain text format. Pulsers encrypt these action and send it to all players, and they send the decryption key to the nearest player at every disclosure timing. A corrupted pulser may send the decryption key earlier or later to a player. It is even possible for a pulser to rewrite a player's action in the plain text format. Players cannot verify how their actions are passed to the other players. Thus, the pulsers constitute a completely trusted third party. The proposed protocol offers better security since time-stamp servers know only hashed actions that are hard to tamper with.

4. Conclusion

By employing time-stamp servers as an infrastructure, the proposed protocol proves a cheat-free protocol for real-time gaming, which achieves minimum latency. Under the assumptions that time-stamp servers honestly issue serially numbered time stamp and that the time-stamp servers are available near each player, the proposed protocol is as fast as and more secure than the fair synchronization protocol, the fastest existing protocol.

A next step is to demonstrate feasibility of the proposed protocol in practice by trial implementation.

Acknowledgments

This work was partially supported by the JSPS Grant-In-Aid no.18300027 and Osamu Miyamoto foundation of the Ibaraki University VBL.

References

[1] P. Laurens, R. F. Paige, P. J. Brooke, and H. Chivers, “A Novel Approach to the Detection of Cheating in Multiplayer

Online Games", 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007), pp. 97-106, July 2007.
[2] N. E. Baughman and B. N. Levine, "Cheat-proof playout for centralized and distributed online games", Proc. IEEE INFOCOM 2001, U.S.A, pp. 104-113, April 2001.
[3] N. E. Baughman, M. Liberation and B. N. Levine, "Cheat-Proof Playout for Centralized and Peer-to-Peer Gaming", IEEE/ACM Transactions on Networking, vol.15, Issue 1, pp. 1-13, February 2007.
[4] H. Lee, S. Lenker, E. Kozlowski, and S. Jamin, "Synchronization and cheat-proofing protocol for multiplayer games", Playing with the Future: Development and Direction in Computer Gaming, April 2002, http://les1.man.ac.uk/cric/gamez/abstracts/lee.html
[5] E. Cronin, B. Filstrup, and S. Jamin, "Cheat-proofing dead reckoned multiplayer games", Proc. Int. Conf. Appl. Devel. Compu. Games(ADCOG) 2003, Hong Kong, January 2003.
[6] B. D. Chen and M. Maheswaran. "A Fair Synchronization Protocol with Cheat Proofing for Decentralized Online Multiplayer Games", Network Computing and Applications, 2004, pp. 372-375, January 2004.
[7] C. Adams, P. Cain, D. Pinkas and R. Zuccherato. "Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)", RFC 3161, August 2001.
[8] Time business accreditation center, "Accreditation program for time-stamping services", 2007, http://www.dekyo.or.jp/tb/english/index.html
[9] T. Nishikawa and S. Matsuoka, "Building Time-Stamp Authority Grid and Basic Experiment", Technical Report of IEICE, vol.107, no.16, pp. 13-18, 2007.

Shunsuke Mogaki was born in Mito, Ibaraki, Japan in 1983. He received his bachelor's degree in engineering in 2007 from Ibaraki University. He is soon finishing his master's degree at the same university. He has been working on security issues in the Internet as well as real world.

Masaru Kamada was born in Hitachi, Ibaraki, Japan in 1962. He received his bachelor's (1984), master's (1986) and doctoral (1988) degrees in engineering from the University of Tsukuba and worked for the same university as a faculty member. In 1992, he joined Ibaraki University where he is currently a professor of computer science. He was a pos-doc fellow at Mathematische Systemtheorie of ETH Zurich in 1993-1995. He served the engineering science society of IEICE as an associate editor of its Japanese transactions (1997-2000), publications secretary (1998-1999) and webmaster (1999). He has been serving *Sampling Theory in Signal and Image Processing* as its secretary since 2003 and *IEEE Transactions on Industrial Electronics* as an associate editor since 2009. He is working also on web-based services, signal processing, and computer forensics. Dr. Kamada is a member of IEEE, IEICE, and EURASIP.

Tatsuhiro Yonekura was born in 1954. He received his B.S. and M.S. from Nagoya University in 1979 and 1981 respectively. He then joined Yamatake Honeywell, working on development of a distributed computer control system till 1990. He received his Ph.D. Degree in Information Science from the Nagoya University in 1991. He joined Ibaraki University in 1991. He is a Professor at Department of Computer and Information Sciences at Ibaraki University. He is a Professional Member of ACM, IEICE (The Inst. of Electronics, Information and Communication Engineers), BMA (Business Model Association) and VRSJ (Virtual Reality Society of Japan). His research interests are in the area of Human Communication on Internet, E-business, E-society and Distributed Virtual Environment.

Shusuke Okamoto was born in Tokyo, Japan, in 1965. He received the B.Eng., M.Eng. and Dr. Eng. degrees from Seikei University, in 1989, 1991 and 1994. In 1994 he joined the University of Electro-Communications as a research associate. From 1997 to 1999, he was an Assistant Professor at the Graduate School of Information Systems, the University of Electro-Communications. From 1999 to 2005, he was an Assistant Professor at the Department of Computer and Information Sciences, Ibaraki University. He is currently an Associate Professor at Faculty of Science and Technology, Seikei University. His current research interests are programming environments, parallel processing and computer architectures. Dr. Okamoto is a member of IEEE, ACM, IPSJ, IEICE and JSSST.

Yasuhiro Ohtaki was born in Yamagata, Japan in 1966. He received his bachelor's degree from the University of Tsukuba in 1989, and his Ph.D (in Engineering) from the same university in 1994. From 1994 to 2001, he has been an research associate at the Department of Computer and Information Sciences at Ibaraki University. Since 2001 he has been a Lecturer. His current research interests include superdistribution and computer forensices. Dr. Ohtaki is a member of IEEE, IEICE and IPSJ.

Mamun Bin Ibne Reaz was born in Bangladesh, in December 1963. He received his B.Sc. and M.Sc. degree in Applied Physics and Electronics, both from University of Rajhashi, Bangladesh, in 1985 and 1986, respectively. He received his D.Eng. degree in 2007 from Ibaraki University, Japan. He is currently a Senior Lecturer in the National University of Malaysia involving in teaching, research and industrial consultation. He is a regular associate of the Abdus Salam International Center for Theoretical Physics since 2008. He has vast research experiences in Norway, Ireland and Malaysia. He has published extensively in the area of IC Design and Biomedical application IC. He is author and co-author of more than 100 research articles in design automation and IC design for biomedical applications. Dr. Mamun is a member of IEEE.